PHOTOGRAPH THIS SHEET

AD A102538

DTIC ACCESSION NUMBER

II

LEVEL

INVENTORY

Maryland Univ., College Park, MD Dept. of Computer Science

International Workshop on High-Level Language Computer Architecture.

DOCUMENT IDENTIFICATION

1 Jul. 79 - 30 Jun. 80 Final Rept.

Contract N00014-79-C-0604 Jun 80

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

DISTRIBUTION STATEMENT

ACCESSION FOR	
NTIS	GRA&I
DTIC	TAB
UNANNOUNCED	
JUSTIFICATION	
BY	
DISTRIBUTION /	
AVAILABILITY CODES	
DIST	AVAIL AND/OR SPECIAL
A	

DISTRIBUTION STAMP

DTIC
ELECTE
S AUG 5 1981 D
D

DATE ACCESSIONED

81 7 17 044

DATE RECEIVED IN DTIC

PHOTOGRAPH THIS SHEET AND RETURN TO DTIC-DDA-2

DTIC FORM 70A OCT 79

DOCUMENT PROCESSING SHEET

N00014-79-C-0604(P0001) June, 1980

Final Report on the International Workshop on High-level Language Computer Architecture

COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

INTERNATIONAL WORKSHOP ON
HIGH-LEVEL LANGUAGE COMPUTER ARCHITECTURE

Date: May 27-28, 1980 (Workshop)
May 26, 1980 (Tutorial)

Location: Bahia Mar Hotel
Fort Lauderdale, Florida

Workshop Committee

Yaohan Chu
University of Maryland

Leonard Haynes
Office of Naval Research

Lee Hoevel
IBM Research Center

George Ligler
Texas Instruments, Inc.

Program Committee

Yaohan Chu (Chairman)
University of Maryland

F. Anceau
Universite de Grenoble

Klaus Berkling
Institut fur Informations-systemforschung

Jack Dennis
Massachusetts Institute of Technology

Keith Doty
University of Florida

Michael J. Flynn
Stanford University

Leonard S. Haynes
Office of Naval Research

Lee Hoevel
IBM Research Center

David K. Hsiao
Ohio State University

J. K. Iliffe
University of London

George T. Ligler
Texas Instruments, Inc.

Glenford J. Meyers
IBM Systems Research Institute

Donald L. Moon
Wright-Patterson Air Force Base

Victor S. Moore
IBM Corporation

Amar Mukhopadhyay
University of Central Florida

Daniel L. Slotnick
University of Illinois

Masahiro Yamamoto
Nippon Electric Co, Ltd.

Tutorial

Keith Doty
University of Florida

Publicity

Lee Hoevel
IBM Research Center

Local Arrangements

Victor Moore
IBM Boca Raton

Publication

Brenda J. Guarnieri
University of Maryland

Treasurer

Jo Ann Thompson
University of Maryland

Registration

Carmen Radelat
University of Maryland

N00014-79-C-0604(P0001) June, 1980

Final Report on the International Workshop on High-level Language Computer Architecture

COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Final Report on The International Workshop on High-Level Language Computer Architecture		5. TYPE OF REPORT & PERIOD COVERED 1 Jul 79 - 30 Jun 80 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Yaohan Chu		8. CONTRACT OR GRANT NUMBER(s) N00014-79-C-0604
9. PERFORMING ORGANIZATION NAME AND ADDRESS University of Maryland Computer Science Department College Park, MD 20742		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Information Systems Program, 437 Arlington, VA 22217		12. REPORT DATE June 1980 13. NUMBER OF PAGES 14
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this report is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

High-level Language,; Computer Archietecture

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The International Workshop on HLLCA was held 26-28 May 1980, at Ft. Lauderdale, FL. This Final Report lists the topics discussed and the participants. A 255-page proceedings was distributed during the Workshop.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Final Report on the International Workshop on High-level Language Computer Architecture

Reported by Yaohan Chu
June 30, 1980

This is the final report for the International Workshop on HLLCA. This Workshop is made possible by the partial support from the ONR. The details of the Workshop are reported below.

1. Summary of the Grant

Title: International Workshop on High-level Language Computer Architecture
Period: 7/1/79 - 6/30/80
Grant no.: N00014-79-C-0604
Grant Amount: $9,860.00
Principal Investigator: Professor Yaohan Chu
Department of Computer Science
University of Maryland
College Park, MD 20742
301-454-4245

2. Workshop

Date: May 26-28, 1980
Location: Fort Lauderdale, FL
No. of Registrants: Technical program: 93 (see Appendix A)
Tutorial program: 69 (see Appendix B)
Programs: See Appendix C
Proceedings: A 255-page proceedings was distributed during the workshop.

3. Organization

The workshop is organized by the Workshop Committee. There are four members on the Workshop Committee; the names are shown in Appendix C.

The technical program is organized by the Program Committee whose chairman is Dr. Yaohan Chu. There are 17 members; the names of these members are also shown in Appendix C. There are 26 papers in 8 sessions in addition to a pannel discussion session. The details of this program are shown in Appendix C.

The tutorial program is organized by Dr. Keith Doty. There are 5 lecturers; each provides a set of notes. The names of the lecturers are shown in Appendix C. The other working members of the workshop are also shown in Appendix C.

The Workshop Committee approved the travel allownaces for 4 international participants who presented a paper as a minimum requirement. These names are shown below.

(1) Professor Yoong-Nien Chen
Department of Compcuers
University of Science and Technology
City of Hefei, Province of Anhui,
The People's Republic of China
Amount: $1,000.

(2) Dr. Masahiro Yamamoto
Central Research Laboratory
Nippon Electric Company, Ltd.
Japan
Amount: $750.

(3) Dr. Esen A. Ozkarahan
Middle East Technical University
Ankara, Turkey
Amount: $500

(4) Mr. J.P. Sansonnet
Universite de Paul Sabatier
Toulouse, France
Amount $500.

4. Next Workshop

The Workshop Committee met on May 28, 1980 and decided to have another workshop because of the attendance beyond expectation. The following are decided.

Date: May 17-20 1982
Location: Fort Lauderdale
Program Chairman: Dr. Lee Hoevel
Program Vice Chairman: Dr. George Ligler

5. International Participation

The Workshop is truly international as there were participants from 12 countries: Brazil, Canada, China, France, Ireland, Italy, Japan, Sweden, Turkey, United Kingdom, U.S.A., West Germany.

6. Official Reports Distribution List

Defense Documentation Center Cameron Station Alexandria, VA 22314	12 copies
Office of Naval Research Arlington, VA 22217	
Information Systems Program (437) Code 200 Code 455 Code 458	2 copies 1 copy 1 copy 1 copy
Office of Naval Research Branch Office, Boston Bldg 114, Section D 666 Summer Street Boston, MA 02210	1 copy
Office of Naval Research Branch Office, Chicago 536 South Clark Street Chicago, IL 60605	1 copy
Office of Naval Research Branch Office, Pasadena 1030 East Green Street Pasadena, CA 91106	1 copy
Naval Research Laboratory Technical Information Division, Code 2627 Washington, D.C. 20375	6 copies
Dr. A. L. Slafkosky Scientific Advisor Commandant of the Marine Corps (Code RD-1) Washington, D.C. 20308	1 copy
Naval Ocean Systems Center Advanced Software Technology Division Code 5200 San Diego, CA 92152	1 copy
Mr. E. H. Gleissner Naval Ship Research & Development Center Computation and Mathematics Department Bethesda, MD 20084	1 copy
Captain Grace M. Hopper (008) Naval Data Automation Command Washington Navy Yard Building 166 Washington, D.C. 20374	1 copy

Appendix A

List of Registrants (Technical Program)

Leon S. Levy
Bell Telephone Laboratories
Whippany, NJ 07981
(201) 386-4955

Hartmut G. Huber
Naval Surface Weapon Center
Box 117
Dahlgren, Va. 22448
(703) 663-8656(office)
(703) 775-7046(home)

N.R. Harris
Stanford University
Computer Systems Lab
Department of Electrical Engineering
Stanford CA 94305
(415) 497-3511

Mary Miller
Bell Laboratories
30W062 Capistrano Ct. Apt. 302
Naperville IL 60540
(312) 462-4269 (office)

John J. Zaloudek
Naval Surface Weapons Center
Dahlgren, Va. 22401
(703) 663-7368

E. Dean Earnest
Burroughs Corporation
25725 Jeronimo Rd.
Mission Viejo, CA 92691
(714) 768-2321

Heinz Schlutter
Gesellschaft fur Mathematik und
Datenverarbeitung, MBH
Postfach 1240
Schloss Birlinhoven
D-5205 St., Augustin 1
Bonn, West Germany

Dr. Klaus Berkling
(same address as Schlutter)

Giorgio Sofi
CSELT, VIA REISS ROMOLI
Torino, Italy 10129
tele. 21691

Tim Merrigan
Floating Point Systems
P.O. Box 23489
Portland, OR. 97223
(503) 641-3151

Reinhard G. Kofer
Siemens AG, ZFE-FL-SAR 112
Otto Hahn Ring 6
8 Muenchen 83 West Germany

Richard C. Fleming
The Aerospace Corperation
M.S. A2/2043
P.O. Box 92957
Los Angeles CA 90009
(213) 648-7098

Dr. G. U. Merckel
IBM Dept. 24k Bldg 032-3
2000 NW 51 Street
Boca Raton FL. 33432
(305) 994-4763

Melvin Hallerman
IBM Dept. 24K bldg 032-3
2000 NW 51 Street
Boca Raton FL. 33432

Kerry V. Richmond
McDonnell Douglas Astronautics Co.
P.O. Box 516
St. Louis, MO. 63166

James D. Mooney
West Virginia University
Dept. STAT. & COMP. Science
Morgantown, WY. 26506
(304) 293-3607

Meir Kaftor M/S B100
Honeywell Information Systems
P.O. Box 6000
Phoenix, AZ. 85005
(602) 866-3381

Nobuyuki Goto
Toshiba Corporation
1 Komukai-Toshiba-cho, Saiwai-ku
Kawasaki, Japan 210
(044) 511-2111

List of Registrants (Technical Program)

Jack B. Dennis
MIT Lab for Computer Science
545 Main Street
Cambridge, MA. 02139
(617) 253-6856

Mou-Shin Yang
Sustems Engineering
6901 W. Sunrise Blvd.
Ft. Lauderdale Fla. 33313
(305) 587-2900 X6236

Gilgert J. Hansen
Texas Instruments
P.O. Bax 222013, MS 3407
Dallas, TX. 75222
(214) 462- 4742

Daniel L. Slotnick
University of Illinois
283 Digital Computer Lab
Dept. of Computer Science
(217) 333-6726

Terry Welch
Sperry Research
100 North Rd.
Sudbury, MA. 01776
(617) 369-4000

Samuel P. Harbison
Carnegie-Mellon University
602A Kelly Ave
Pittsburgh, Pa. 15221
(412) 731- 1472

Charles W. Flink II
Naval Surface Weapon Center
K-74
Dahlgren, Va. 22401
(703) 663-7517

Bill Kwinn
Hewlett Packard
3404 E. Marmony Road
Fort Collins, CO 80525
(303) 226-3800 X3242

Jaishanker Menon
Dept of Computer Science
Ohio State University
Columbus Ohio 43210
(614) 422-5813

Harvey G. Cragon
Texas Instruments, Inc.
P.O.Box 225012
Dallas, TX 75265
(214) 238-3023

Leon I. Maissel
IBM Corp
Dept. C14, Bldg 704,
P.O.Box 390
Poughkeepsie, NY 12602
(914) 463-2301

Raymond L. Phoenix
IBM Corp
Dept. C14, Bldg 704,
P.O.Box 390
Poughkeepsie, NY 12602
(914) 463-5445

Zvi Weiss
IBM Research Center
Yorktown Heights, NY 10598
(914) 962-7036

Richard Ramseyer
Honeywell SRC Research
2600 Ridgway Pkwy, MN17-2352
Minneapolis, MN 55413
( ) 378-5023

Tetsuo Ida
Institute of Physical & Chem. Res.
2-1, Hirosawa,
Wako-shi, Saitama 351
Japan

Greg Bettice
Naval Vaionics Center
8125 Harrison Drive
Lawrence, IN 46226
(317) 353-3226

Roger R. Bate
Texas Instruments, Inc.
P.O.Box 222013, M/S 3407
Dallas, TX 75222
(214) 462-4790

Ron Rutledge
DOT/TSC, P.O.Box 53
Kendall Square
Cambridge, MA 02142
(617) 494-2038

List of Registrants (Technical Program)

Gerhard Herrscher
LITEF
Loerracher Strasse 18
7800 Freiburg
West Germany
0761-4901212

A. Speckhard
Aerospace Corperation
2350 E. El Segundo Blvd.
El Segundo CA 90245
(213) 648-7067

John Francis
Sanders Associates, Inc.
95 Canal Street
Nashua NH 03060
(603) 885-3746

Paula Bernstein
Bell Laboratories
Warrenville-Naperville Rds.
Naperville IL 60540
(312) 462-2898

R.F. Hobson
Simon Fraser University
S.F. University
Computer Science Department
Burnaby British Columbia VSAIS6
(604) 291-4277

Dr. Werner Kluge
GMD/ISF
Postfach 1240
SchloB Birlinghoven
West Germany

Malcolm Muir
Datamedix, Inc.
555 Hillsboro Plaza
Deerfield Beach FL 33441
(305) 428-4526

Ronald L. Engelbrecht
NCR Corp. - E&M-Wichita
3718 N. Rock Road
Wichita KS 67218
(316) 688-8646

Dr. F.J. Burkowski
Computer Science Department
University of Manitoba
Room 545 Machray Hall
Winnipeg Manitoba, Canada R3T 2N2
(204) 47408313

Allen Baum
Hewlett-Packard
HPL/CRL
1501 Page Mill Rd.
Palo Alto CA 94304
857-8776

Keiji Kuwahara
Nikkei-McGraw-Hill
2-1-2 Uchikanda, Chiyoda-ku
Tokyo Japan
(03) 256-1561

Y. El-ziq
Honeywell
Honeywell Plaza
Minneapolis Minnisota 55408

David E. Heinen
Tektronix, Inc.
P.O. Box 500 DS 63-311
Beaverton OR 97077
(503) 682-3411 x3845

Lawrence Katz
Tektronix, Inc.
P.O. Box 500 DS 63-311
Beaverton OR 97077
(503) 682-3411 x3081

R. Curtis
Canisius College
2011 Main Street
Buffalo NY 14208
(716) 831-7000

John Bowles
NCR Corporation
3325 Platt Springs Rd.
C. Columbia SC 29169
(803) 796-9250 x524

David M. Abrahamson
Department of Computer Science
Trinity College
Dublin 2 Ireland
772941 Ext. 1765

Hugh L. Applewhite
Honeywell 17-2352
2600 Ridgway N.E.
Minneapolis, MN 55413
(612) 378-4510

List of Registrants (Technical Program)

David K. Hsiao
Ohio State University
Department of Computer Science
Columbus Ohio 43210
(614) 422-5813

M. Tsuchiya
TRW DSSG R2/2036
One Space Park
Redondo Beach CA 90278
(213) 535-0580

Dr. William D. Murray
University of Colorado
1100 14th Street
Denver CO 80202
(303) 629-2872

Bantwal R. Rau
Coordinated Science Laboratories
University of Illinois
Urbana IL 61801
(217) 333-7146

Leonard Haynes
Office of Naval Research
Arlington VA 22217
696-4302

Yaohan Chu
University of Maryland
Department of Computer Science
College Park, Maryland 20742
(301) 454-4245

Krishna M. Kavipurapu
Southern Methodist University
Department of Computer Science
Dallas TX 75275
(214) 692-3095

Barry C. Goldstein
IBM T.J.Watson Research
24 Glen Terrace
Chappaqua NY 10514
(914) 945-2693 (office)

David A. Patterson
University of California
Electrical Engineering and
Computer Sciences
Computer Science Division
Berkeley, California 94720

G.T. Ligler
Burroughs Corp.
P.O. Box 517
Paoli, PA 19301
215-648-3248

Robert F. Cmelik
Bell Laboratories
Room 7D-414
600 Mountain Ave.
Murray Hill NJ 07974
(201) 582-5797

David R. Ditzel
Bell Laboratories
2C-523
Murray Hill NJ 07974
(201) 582-3655

Thomas A. Almy
Tektronix, Inc. M/S 50-384
Box 500
Beaverton OR 97077
644-0161 x6056

Herman Hartig
Universitat Karlsruhe
Institut fur Informatik IV
75 Karlsruhe I
Postfach 6380
Zirkel Nr.:2
W-Germany

Kemal Oflazer
M.E.T.U.
Ankara, Turkey

John Peterson
Universtiy of Colorado
2845 S. Gilpin
Denver CO 80210
(303) 629-2872

Bernard Lecussan
36 Impasse st. Felix
31400 Toulouse, France

Jean-Paul Sansonnet
15 Rue ctre Midi Bat. 1
31400 Toulouse, France

Lars-Erik Thorelli
Royal Institute of Technology
S-100yy Stockhom, Sweden

Goran Bage
Royal Institute of Technology
LM Erricsson
S-12625 Stockholm, Sweden

Dennis A. Roberson
IBM - Boca Raton
Boca Raton FL 33432

List of Registrants (Technical Program)

Dick Coun
Fairchild Camera & Instrument Corp.
464 Ellis Street
Mt. View, CA 94040
(415) 962-2337
962-4523

Tich T. Dao
Fairchild Camera & Instruments Corp.
464 Ellis Street
Mt. View, CA 94040
(415) 962-7532
962-4523

Mark T. Michael
US Air Force, Avionics Lab
WPAFB OH 45433
(513) 255-4920

Dr. Esen A. Ozkarahan
METU
BMB-METU (ODTH)
Ankara Turkey

J. K. Illiffe
International Computers Ltd.
37 Western Road
London N2 9JB
England

Leonard J. Disch
IEEE
Flaun ser Strausse 17
7801 Stegen/Eschbach
West Germany

Masahiro Yamamoto
NEC
4-1-1 Miyazaki Takatsu-Ku
Kawasaki, Tokyo
(044) 855-1111

Lonny R. Lebahn
Sperry Univac
7807 64th St. N
Pine Springs, MN 55109
(612) 631-6901

Lucas A. Moscato
Escola Politecnica da USP, Dept. Engenharia de Eletricidade Cidade Universitaria-C. Postal 11.455
05508 - Sao Paulo - SP, BRAZIL

Jack Quanstrom
IBM Corp.
P.O.Box 1328,Dept.24k/032-3
Boca Raton, FL. 33432
994-4770

Glenford J. Myers
IBM Systems Research Inst.
205 E 42nd Street
New York, NY 10017
(212) 983-7250

M. Durrieu
CERA - CERT
Z Ar E. Berlin BP 4025
Toulouse, France (31055)
(61) 25 21 88

Pong-sheng Wang
Ohio State University
2036 Neil Avenue
Columbus, Ohio 43202
(614) 422-8039

Dipl.-Ing Rudiger Strelow
Siemens AG-Bereich Systemtechnische Entwicklung
Steuerungs-und Informationssyteme
Gunter-Scharowsky-Strabe 2
(09131) 7-6923

William M. Cooper
Softech, Inc.
4140 Linden Avenue
Dayton, Ohio 45432
513-253-1522

Robert Kopac
IBM-Poughkeepsie, N.Y.
39 High Acres Drive
Poughkeepsie, N.Y. 12603
914-452-2049

Martin Freeman
Bell Laboratories
Whippany Road
Whippany, N.J. 07981
201-455-1895

Malcolm Harrison
N.Y.U.
251 Mercer St.
N.Y., N.Y. 10012
212-460-7287

Appendix B

List of Registrants (Tutorial Program)

Herbert Schorr
IBM T. J. Watson Research Center
P.O. Box 218
Yorktown Heights, N.Y. 10598
(914) 945-1285

Hartmut G. Huber
Naval Surface Weapon Center
Box 117
Dahlgren, Va. 22448
(703) 663-8656(office)
(703) 775-7046(home)

Richard C. Fleming
The Aerospace Corporation
M.S. A2/2043
P.O. Box 92957
Los Angeles, CA 90009

Joseph M. Herko
IBM Corporation
P.O. Box 1328, Dept. 25T 032-1
Boca Raton, Florida 33432
(305) 994-3458

John J. Zaloudek
Naval Surface Weapons Center
Dahlgren, Va. 22401
(703) 663-7368

E. Dean Earnest
Burroughs Corporation
25725 Jeronimo Rd.
Mission Viejo, CA 92691
(714) 768-2321

Heinz Schlutter
Gesellschaft fur Mathematik und
Datenverarbeitung, MBH
Postfach 1240
Schloss Birlinhoven
D-5205 St., Augustin 1
Bonn, West Germany

Dr. Klaus Berkling
(same address as Schlutter)

Tim Merrigan
Floating Point Systems
P.O. Box 23489
Portland, OR. 97223
(503) 641-3151

Reinhard G. Kofer
Siemens AG, ZFE-FL-SAR 112
Otto Hahn Ring 6
8 Muenchen 83 West Germany

Mr. Lucas Moscato
No Address
Country: Brazil

Dr. G. U. Merckel
IBM Dept. 24k Bldg 032-3
2000 NW 51 Street
Boca Raton FL. 33432
(305) 994-4763

Melvin Hallerman
IBM Dept. 24K bldg 032-3
2000 NW 51 Street
Boca Raton FL. 33432

Kerry V. Richmond
McDonnell Douglas Astronautics Co.
P.O. Box 516
St. Louis, MO. 63166

James D. Mooney
West Virginia University
Dept. STAT. & COMP. Science
Morgantown, WY. 26506
(304) 293-3607

Meir Kaftor M/S B100
Honeywell Information Systems
P.O. Box 6000
Phoenix, AZ. 85005
(602) 866-3381

Nobuyuki Goto
Toshiba Corporation
I Komukai-Toshiba-cho, Saiwai-ku
Kawasaki, Japan 210
(044) 511-2111

Tutorial Program

Gerhard Herrscher
LITEF
Loerracher Strasse 18
7800 Freiburg
West Germany
0761-4901212

A. Speckhard
Aerospace Corperation
2350 E. El Segundo Blvd.
El Segundo CA 90245
(213) 648-7067

John Francis
Sanders Associates, Inc.
95 Canal Street
Nashua NH 03060
(603) 885-3746

Paula Bernstein
Bell Laboratories
Warrenville-Naperville Rds.
Naperville IL 60540
(312) 462-2898

R.F. Hobson
Simon Fraser University
S.F. University
Computer Science Department
Burnaby British Columbia VSAIS6
(604) 291-4277

Dr. Werner Kluge
GMD/ISF
Postfach 1240
SchloB Birlinghoven
West Germany

Malcolm Muir
Datamedix, Inc.
555 Hillsboro Plaza
Deerfield Beach FL 33441
(305) 428-4526

Ronald L. Engelbrecht
NCR Corp. - E&M-Wichita
3718 N. Rock Road
Wichita KS 67218
(316) 688-8646

Dr. F.J. Burkowski
Computer Science Department
University of Manitoba
Room 545 Machray Hall
Winnipeg Manitoba, Canada R3T 2N2
(204) 47408313

Allen Brown
Hewlett-Packard
HPL/CRL
1501 Page Mill Rd.
Palo Alto CA 94304
857-8776

Keiji Kuwahara
Nikkei-McGraw-Hill
2-1-2 Uchikanda, Chiyoda-ku
Tokyo Japan
(03) 256-1561

Y. El-zig
Honeywell
Honeywell Plaza
Minneapolis Minnisota 55408

David E. Heinen
Tektronix, Inc.
P.O. Box 500 DS 63-311
Beaverton OR 97077
(503) 682-3411 x3845

Lawrence Katz
Tektronix, Inc.
P.O. Box 500 DS 63-311
Beaverton OR 97077
(503) 682-3411 x3081

R. Curtis
Canisius College
2011 Main Street
Buffalo NY 14208
(716) 831-7000

John Bowles
NCR Corporation
3325 Platt Springs Rd.
C. Columbia SC 29169
(803) 796-9250 x524

David M. Abrahamson
Department of Computer Science
Trinity College
Dublin 2 Ireland
772941 Ext. 1765

Hugh L. Applewhite
Honeywell 17-2352
2600 Ridgway N.E.
Minneapolis, MN 55413
(612) 378-4510

Tutorial Program

Leon I. Maissel
IBM Corp
Dept. C14, Bldg 704,
P.O.Box 390
Poughkeepsie, NY 12602
(914) 463-2301

Raymond L. Phoenix
IBM Corp
Dept. C14, Bldg 704,
P.O.Box 390
Poughkeepsie, NY 12602
(914) 463-5445

Zvi Weiss
IBM Research Center
Yorktown Heights, NY 10598
(914) 962-7036

Richard Ramseyer
Honeywell SRC Research
2600 Ridgway Pkwy, MN17-2352
Minneapolis, MN 55413
( ) 378-5023

Tetsuo Ida
Institute of Physical & Chem. Res.
2-1, Hirosawa,
Wako-shi, Saitama 351
Japan

Greg Bettice
Naval Vaionics Center
8125 Harrison Drive
Lawrence, IN 46226
(317) 353-3226

Roger R. Bate
Texas Instruments, Inc.
P.O.Box 222013, M/S 3407
Dallas, TX 75222
(214) 462-4790

Ron Rutledge
DOT/TSC, P.O.Box 53
Kendall Square
Cambridge, MA 02142
(617) 494-2038

Robert F. Cmelik
Bell Laboratories
Room 7D-414
600 Mountain Ave.
Murray Hill NJ 07974
(201) 582-5797

David R. Ditzel
Bell Laboratories
2C-523
Murray Hill NJ 07974
(201) 582-3655

Thomas A. Almy
Tektronix, Inc. M/S 50-384
Box 500
Beaverton OR 97077
644-0161 x6056

Herman Hartig
Universitat Karlsruhe
Institut fur Informatik IV
75 Karlsruhe I
Postfach 6380
Zirkel Nr.:2
W-Germany

Mary Miller
Bell Laboratories
30W062 Capistrano Ct. #302
Naperville, Ill. 60540
(312) 462-4269

John Peterson
University of Colorado
2845 S. Gilpin
Denver CO 80210
(303) 629-2872

Bernard Lecussan
36 Impasse St. Felix
31400 Toulouse, France

Jean-Paul Sansonnet
15 Rue ctre Midi Bat.1
31400 Toulouse, France

Tutorial Program

David A. Patterson
University of California
Electrical Engineering and
and Computer Sciences
Computer Science Division
Berkeley, CA 94720

Lars-Erik Thorelli
Royal Institute of Technology
S-10044 Stockholm Sweden

N.G. Frank Thoma
IBM
4686 NW 2nd Ct.
Boca Raton FL 33431
(305) 368-4676

Joseph C. Rhodes, Jr.
IBM Corporation
P.O. Box 1328
Boca Raton FL 33432
(305) 994-7654

Goran Bage
LM Ericsson
S-12625 Stockholm, Sweden

Peter Klambatsen
IBM Corporation
2000 NW 51st Street
P.O. Box 1328
Boca Raton, FL 33064
994-5098

Moises Cases
IBM-GSD
Yamato Road
Boca Raton, FL
994-7992

Dick Conn
Fairchild Camera &
Instrument Corp.
464 Ellis Street
Mt. View, CA 94040
(415) 962-2337

Jack Quanstrom
IBM Corporation
P.O. Box 1328
Dept. 24K/032-3
Boca Raton, FL 33432
994-4770

Tich T. Dao M/S 17-5904
Fairchild Camera &
Instrument Corp.
464 Ellis Street
Mt. View, CA 94040
(415) 962-7532

Mark T. Michael
US Air Force, Avionics Laboratory
WPAFB, OH 45433
(513) 255-4920

(3) "SWARD - A Software-oriented Architecture"
Glen Myers
IBM Systems Research Institute

(4) "Considerations in Operating Systems Design for Multiprocessor Structures"
H. Lagrin
IBM Systems Research Institute
Barry C. Goldstein
IBM Research Center

12:00 - 2:00 pm Lunch

2:00 - 3:30 pm Session H: Functional Programming Language Architecture

Chairman: Dr. Klaus Berkling
Gesellschaft fur Mathematik und Datenverarbeitung
mbH Bonn, West Germany

(1) "An Architecture for Direct Execution of Reduction Languages"
W. Kluge & H. Schlutter
Gesellschaft fur Mathematik and Datenverarbeitung
mbH Bonn, West Germany

(2) "An Expression Oriented Editor for Language with a Constructor Syntax
Gesellschaft fur Mathematik and Datenverarbeitung
mbH Bonn, West Germany

(3) "Parsi A Computer Employing Functional Programming Systems"
J. C. Peterson & W. D. Murray
University of Colorado at Denver

3:30 - 3:45 pm Coffee Break

3:45 - 5:15 pm Session I: High-level Architecture III
Chairman: John K. Iliffe
University of London

(1) "On a Document Preparation Machine"
Martin Freeman & Leon S. Levy
Bell Laboratories, Whippany

(2) "A High-level Architecture for a Text Scanning Processor"
F. J. Berukowski
University of Manitoba
Canada

(3) "A COBOL Machine Design and Evaluation"
M. Yamamoto, R. Nakazaki, M. Yohota, & M. Umemura
Nippon Electric Co., Ltd.
Japan

WORKSHOP COMMITTEE

Yaohan Chu
University of Maryland

Leonard Haynes
Office of Naval Research

Lee Hoevel
IBM Research Center

George Ligler
Texas Instruments, Inc

PROGRAM COMMITTEE

Yaohan Chu (Chairman)
University of Maryland
U.S.A.

F. Anceau
Universite de Grenoble
France

Klaus Berkling
Institut fur Informations-systemforschung
West Germany

Jack Dennis
Massachusetts Institute of Technology
U.S.A

Keith Doty
University of Florida
U.S.A.

Michael J. Flynn
Stanford University
U.S.A.

Leonard S. Haynes
Office of Naval Research
U.S.A.

Lee Hoevel
IBM Research Center
U.S.A.

David K. Haiao
Ohio State University
U.S.A.

J. K. Iliffe
University of London
England

George T. Ligler
Texas Instruments, Inc.
U.S.A.

Glenford J. Meyers
IBM Systems Research Institute
U.S.A.

Donald L. Moon
Wright-Patterson Air Force Base
U.S.A.

Victor S. Moore
IBM Corporation
U.S.A.

Amar Mukhopadhyay
University of Central Florida
U.S.A.

Daniel L. Slotnick
University of Illinois
U.S.A.

Masahiro Yamamoto
Nippon Electric Co., Ltd.
Japan

TUTORIAL

Keith Doty
University of Florida

PUBLICITY

Lee Hoevel
IBM Research Center

LOCAL ARRANGEMENTS

Victor Moore
IBM Boca Raton

PUBLICATION

Brenda J. Guarnieri
University of Maryland

TREASURER

Jo Ann Thompson
University of Maryland

Appendix C

INTERNATIONAL WORKSHOP ON HIGH-LEVEL LANGUAGE COMPUTER ARCHITECTURE

Date: May 26, 1980 (Tutorial)
May 27-28, 1980 (Workshop)

Location: Bahia Mar Hotel
Fort Lauderdale, Florida, 33316
305-764-2233

MONDAY, MAY 26, 1980 (TUTORIAL)

8:45 - 9:00 am Tutorial Chairman Dr. Keith L. Doty

9:00 - 10:15 am Lecturer: Dr. John K. Iliffe
University of London

Topic: Micro systems support for high-level languages
This tutorial discusses system interfaces particularly intended for use with variable microprograms. In this context, the problems of security, expansion, maintenance, and so on whose solution is normally taken for granted, can easily undermine progress. A set of support functions based on experience with several language oriented designs will be outlined.

10:15 - 10:45 am Coffee Break

10:45 - 12:00 am Lecturer: Dr. Yaohan Chu
University of Maryland

Topic: Direct Execution Computer Design
This tutorial introduces the design of high-level language direct execution computer. It presents basic concepts. It describes in great details how such a direct-execution computer works. It outlines a design procedure and illustrates with a simple example.

12:00 - 2:00 pm Lunch

2:00 - 3:15 pm Lecturer: Dr. Michael Flynn
Stanford University

Topic: Ideal Language Machines
This tutorial presents ideal computer architecture which can be defined as the representation originally used to describe a higher level language program. A canonic interpretive form (CIF) or measure of a high level language program is developed. The CIF or ideal program representation is then compared using Whetstone benchmark in its characteristics to several contemporary architectural approaches

3:15 - 3:30 pm Coffee Break

3:30 - 4:45 pm Lecturer: Dr. Lee Hoevel
IBM Research Center

Topic: Directly Executed Language Design
This tutorial presents a reasonably comprehensive theory of synthesizing directly executed language (DEL). It presents a complete implementation of a simple version of FORTRAN. This DEL for FORTRAN called DELTRAN comes close to achieving the ideal program measures.

4:45 - 6:00 pm Lecturer: Mr. D. A. Roberson
IBM Boca Raton

Topic: IBM Low-end High-level Language Machines

This tutorial focuses on the architecture of the IBM Low-end High-level Language machines, 5100/5110/5120. It includes a description of the philosophy for this class of machines as well as a description of the specific implementation approach. It also attempts to describe the development process of this class of machines in IBM which was used to produce the machines.

TUESDAY, MAY 27, 1980

8:15 - 8:30 am Program Chairman Dr. Yaohan Chu

8:30 - 10:00 am Session A: High level Architecture I

Chairman: Dr. Victor Moore
IBM Boca Raton

(1) "The Architecture of a Parallel Executing High-level Language Computer"
Ming T. Liu & F. S. Wang
Ohio State University

(2) "Direct-Execution High-level-language Fortran Computer"
Y. N. Chen, K. L. Chen, & K. C. Huang
China University of Science and Technology
People's Republic of China

(3) "A JOVIAL Direct Execution Machine
Yaohan Chu
University of Maryland

10:00 - 10:30 am Coffee Break

10:30 - 12:00 pm Session B: Directly Executable Languages

Chairman: Dr. Michael Flynn
Stanford University

(1) "Quest for an 'Ideal' Machine Language"
Krishna M. Kavipurapu & Harvey Cragon
Southern Methodist University

(2) "A Directly Executable Language for Bit Slice Microprocessor Implementation"
N. R. Harris
Stanford University

(3) "Partial Evaluation of a High-level Architecture"
Lars-Erik Thorelli
Royal Institute of Technology
Sweden

(4) "Directly Interpretable Language Design for High-level Language Support"
B. R. Rau & P. Bose
University of Illinois at Urbana

12:00 - 2:00 pm Lunch

2:00 - 3:30 pm Session C: Issues and Perspectives

Chairman: Dr. George Ligler
Texas Instruments, Inc.

(1) "Twenty Years of Burroughs High-level Language Machines"
Dean Earnest
Burroughs Corp

(2) "A Survey of High-level Language Machines in Japan"
Masahiro Yamamoto
Nippon Electric Co., Ltd.

(3) "Reflections on a High-level Language Computer System or Parting Thoughts on the SYMBOL Project"
David R. Ditzel & William A. Kwinn
Bell Laboratories, Murray Hill

(4) "A Case Against High-level Language Computer Architecture"
H. C. Cragon
Texas Instruments, Inc.

3:30 - 3:45 pm Coffee Break

3:45 - 5:15 pm Session D: Data Base Architecture

Chairman: Dr. Leonard Haynes
Office of Naval Research

(1) "Design Issues of High-level Language Database Computers"
David K. Hsiao
Ohio State University

(2) "Hashing Hardware and It's Application to Symbol Manipulation"
Tetsuo Ida
Institute of Research
Japan

(3) "RAP.3 - A multi-microprocessor Cell Architecture for the RAP Database Machine"
Esen A. Ozkarahan & K. Ollazer
Middle East Technical University
Ankara, Turkey
K. C. Smith
University of Toronto

TUESDAY EVENING, MAY 27, 1980

8:00 - 10:00 am Panel Session E: Future Impact of High-level Architecture

Chairman: Dr. William A Wulf
Carnegie-Mellon University

Panelists: Dr. Herbert Schorr
IBM Research Center
Mr. Harvey C. Cragon
Texas Instruments, Inc.
Dr. Leonard Haynes
Office of Naval Research
Dr. Jack Dennis
Massachusetts Institute of Technology
Mr. Dean Earnest
Burroughs Corporation

WEDNESDAY, MAY 28, 1980

8:30 - 10:00 am Session F: High Level Architecture II

Chairman: Mr. Masahiro Yamamoto
Nippon Electric Co., Ltd.

(1) "Architecture of a Multi-language Processor Based on List-Structured DEL's"
J. P. Sansonnet, M. Castan, & C. Percebois
Universite Paul Sabatier
France

(2) "High level Architecture for a Real Time Language LTR"
C. Durrieu, B. Froment, F. Cuaziti, B. Lecussal, J. Romain, D. Vidal, & J. Vuerier
Universite Paul Sabatier
France

(3) "An Architecure for the Dynamic Optimization of High level Language Programs"
S. P. Harbison & Wm. A. Wulf
Carnegie-Mellon University

10:00 - 12:00 pm Session G: System-Oriented Architecture

Chairman: Dr. Lee Hoevel
IBM Research Center

(1) "Architectural Support for Abstraction"
J. K. Iliffe
University of London

(2) "Hierarchial Finite State Machines as a Structure for Input-Output Systems"
H. L. Applewhite
Honeywell Research Center

PROCEEDINGS

OF THE

INTERNATIONAL WORKSHOP

ON

HIGH-LEVEL LANGUAGE

COMPUTER ARCHITECTURE

MAY 26-28, 1980

FORT LAUDERDALE, FLORIDA

SPONSORED BY
THE DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

Acknowledgement

The International Workshop on High-level Language Computer Architecture acknowledges the partial support which it received from the Office of Naval Research.

University of Maryland
Department of Computer Science
College Park, Maryland 20742

A

Department of Computer Science
University of Maryland
College Park, Maryland 20742
U.S.A.
(301) 454-2002

International Workshop on High-level Language Computer Architecture

Table of Contents

Papers for Publication Only

THE ARCHITECTURE OF A PARALLEL EXECUTION HIGH-LEVEL LANGUAGE COMPUTER *

Pong-shang Wang and Ming T. Liu

Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210

Abstract

This paper presents an internal language for a high-level language compter to facilitate parallel execution of arithmetic expressions and concurrent statements and to perform try-ahead operations for IF, WHILE, and REPEAT statements. The architecture of such a computer is also described, which consists of multiple independent processors for language processing and parallel computation. The increase in speed is achieved by parallel execution, by try-ahead processing, and by the pipeline effect created by the independent processors simultaneously performing various tasks. An algorithm that translates an arithmetic expression into the internal language form is also included in the Appendix.

I. Introduction

In the area of high-level computer architecture, various machine organizations have been proposed with features to increase the program processing speed [1] [2]. These designs include independent processors to perform various tasks in language translation and execution, such as the lexical processor, syntactic processor, semantic processor, arithmetic processor, etc. These processors operate simultaneously and asynchronously, and create a pipeline effect in the whole system. The concurrency among these processors results in the speed increase in language translation and execution.

In this paper, however, we look into another possibility of gaining speed in high-level language computers, namely, the parallel execution of arithmetic expressions and concurrent statements, and the try-ahead processing of statements involving conditions, such as IF, WHILE, and REPEAT. The scheme that we use here calls for an indirect-execution architecture which uses an internal language and is of type 3 according to Chu's classification [3]. Source programs are translated into the internal representation, which is then interpreted by the machine hardware. We will describe first the features in the internal language that make parallel and try-ahead operations possible, and then the computer organization for carrying out these operations. We will discuss only the features in the internal language that are relevant to parallel execution and try-ahead processing, and ignore others such as identifiers, labels, etc., since they are immaterial to the purpose of this paper and they can be found in other papers, e.g. [1] [4]. The syntax and semantics of the high-level language constructs are the same as those in PASCAL.

In Section 2.1, we first briefly describe the notion of Parallel Execution Strings (PES) for executing arithmetic expressions in parallel, and then propose a linear representation scheme as the internal language for a high-level computer. An algorithm which translates an arithmetic expression into the internal language form is included in the Appendix. In Section 2.2, we present a method of representing a concurrent statement in the internal language so that it can be executed concurrently. In Sections 2.3 through 2.5, we describe the representation of IF statements, WHILE statements, and REPEAT statements in the internal language for try-ahead processing. The representation allows the possible paths in a statement involving a condition to be executed even before the evaluation of the condition is completed. Finally, a high-level computer organization is presented in Section III, which includes independent processors for language processing, and multiple Semantic Processors and PES Access Processors for parallel computations. In the computer organization, each execution stream is accessed and executed by a PES Access Processor and a Semantic Processor. Each Semantic Processor has its own Arithmetic Processor and Local Storage for concurrent processing and try-ahead processing.

II. Internal Language Constructs

2.1 Arithmetic Expressions for Parallel Execution

A scheme for decomposing arithmetic expressions for parallel execution, called the Parallel Execution String (PES), has been proposed in [5] [6]. It can be summarized as follows.

* Research reported herein was supported in part by NSF-MCS-77-23496.

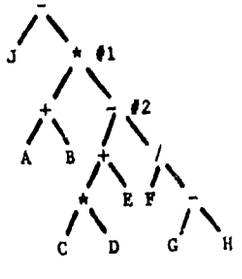
Definition

In an expression tree, an operator node is called

type 1 -- if all of its operands are variables or constants;
type 2 — if exactly one of its operands is an operator; and
type 3 — if it is a binary operator and both of its operands are operators.

Consider an expression in its tree representation. Those operator nodes, the operands of which are variables or constants (i.e., type 1), will be the starting points of the parallel execution strings. Beginning at the starting points, these strings are executed in the direction toward the root node, each of which can be simultaneously executed by an independent processor. Each processor executes the type 2 operators in a string one by one at its maximum speed without waiting. At an operator node where two strings meet (i.e., type 3), the processor which reaches this node first will deposit the partial result it obtains thus far into a temporary storage and then stop, whereas the other processor which reaches this node later will execute the operation at the merging node and continue to execute the remaining string. For example, the expression tree in Figure 1 has three type 1 nodes: A+B, C*D, and G-H; and hence there are three parallel execution strings. The two type 3 nodes in Figure 1 are labeled as #1 and #2, respectively. Note that the number of type 3 nodes is always one less than the number of type 1 nodes.

The expression J-(A+B)*(C*D+E-F/(G-H)) can be represented as a tree:



It can be translated into the internal language as:

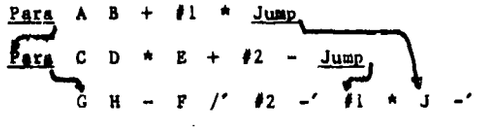


Fig.1 Example of Translating an Expression into the Internal Language

To implement this concept in a high-level language computer, we have to devise a linear representation for the parallel execution strings in an expression tree and use it as the internal language for the high-level language computer. With this internal language, the entry points of the strings are chained as a linked list by pointers called Parallel Pointers. For the operator where two strings meet, one of its two operands is the result of the previous operation in the processor and hence need not be specified, and the other operand is represented by #i, where i is a unique number identifying a temporary storage for the partial result obtained by the processor executing the other string. The first of the two merging strings has a Jump Pointer following the merging point operator and pointing to the location that immediately follows the merging point operator in the second string.

To eliminate the need of a stack during the execution of arithmetic expressions, the ordering of operands will be reversed in the following situation: when the result of the previous operator is the second operand of the current operator, the first operand will appear as the second operand in this representation. Thus, if the operator is non-commutative, it will be marked with an apostrophe following the operator to indicate that the ordering of its operands is reversed.

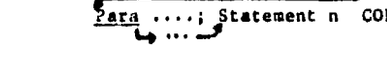
Figure 1 is an example of representing an arithmetic expression in the internal language. In Figure 1, Para represents a Parallel Pointer, and Jump a Jump Pointer. When a PES Access Processor executes a Parallel Pointer (see Section III and Figure 3,) it will put the pointer value into one of the Entry Point Registers so that the next string can be chosen for execution as soon as another PES Access Processor becomes free.

2.2 Concurrent Statements

A concurrent statement [7] is a set of statements enclosed by a header COBEGIN and a trailer COEND; for example,

```
COBEGIN
        Statement 1;
        statement 2;
            .
            .
            .
        Statement n
COEND
```

The statements in a concurrent statement can be executed simultaneously. A flowchart of the above concurrent statement is shown in Figure 2. To execute the concurrent statement, it will be translated into the internal language as follows:



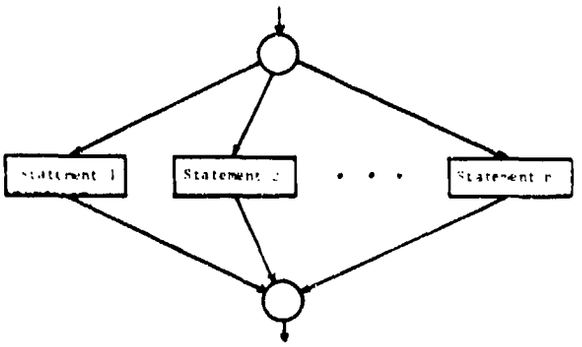


Figure 2 Flowchart of a Concurrent Statement

The processor that executes Statement n will execute COEND. The effect of executing COEND is that the processor will halt its execution temporarily until all the other processors become free.

The semicolons in a concurrent statement will be preserved in the internal language. A semicolon indicates the end of a simple statement in a concurrent statement and hence makes the processor which is executing the simple statement free.

2.3 IF Statements

IF statements will be processed with a try-ahead method. The following IF statement

IF condition THEN statement 1 ELSE statement 2;

will be translated into the internal language as follows:

Para condition IF Para THEN statement 1 THENEND
Jump ELSE statement 2 ELSEND

The processor which starts executing the IF statement will set up the entry to the THEN clause for a second processor, which in turn sets up the entry to the ELSE clause for a third processor. While the first processor is evaluating the conditional expression, both the statement 1 and the statement 2 are being executed simultaneously. However, any environment changes resulting from the execution of the statement 1 and the statement 2 are kept in the local storage of the second and the third processors, respectively, and will have no effect either on the execution of the other or on the evaluation of the conditional expression.

Executing the symbols "THEN" and "ELSE" causes the processor to enter the THEN state and the ELSE state, respectively. When the THEN state processor executes the symbol "THENEND", or when the ELSE state processor executes the symbol "ELSEND", the processor will halt its execution in the WAIT state. However, "THENEND" and "ELSEND" will have no effect on a processor which is in the normal mode of operation.

When the first processor executes the symbol "IF", it interrupts both the second and the third processors. Depending upon the result of the conditional expression, it makes one of the two processors free immediately and discards any computation the processor has done; any environment changes made by the other processor are copied into the main storage and the processor becomes free. When that is done, the first processor resumes execution from where the latter processor was interrupted.

2.4 WHILE Statements

WHILE statements and REPEAT statements will be processed with the try-ahead method similar to that for IF statements. However, only the repetitive path will be tried in advance.

The WHILE statement

WHILE condition DO statement 1;

will be translated as:

Para condition WHILE WHILEDO statement1 WHILEND Jump

The processor which executes the conditional expression sets up the entry to the WHILEDO path for a second processor. The conditional expression and the the WHILEDO path are then executed simultaneously.

Executing the symbol "WHILEDO" forces the second processor to enter the WHILEDO state. The environment changes made by a WHILEDO state processor do not affect the main storage and are only kept in the local storage of the processor. A WHILEDO state processor will halt its execution in the WAIT state when it executes the symbol "WHILEND." However, the "WHILEND" will have no effect on a processor which is in the normal mode of operation.

When the first processor executes the symbol "WHILE," it interrupts the second processor. If the result of the conditional expression is FALSE, the second processor becomes free immediately and everything in its local storage will not be used. The first processor then follows the WHILE pointer to execute the next statement.

If the result is TRUE, the environment changes stored in the local storage of the second processor will be copied into the main storage, and the second processor becomes free. The first processor then resumes execution from where the second processor was interrupted.

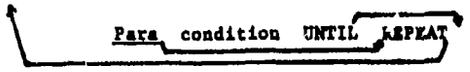
2.5 REPEAT Statements

The REPEAT statement

```
REPEAT
        statement 1;
        statement 2;
            .
            .
            .
        statement n
UNTIL condition;
```

will be translated as:

statement 1 statement2 ... statement n REPEATEND



The processor which executes the conditional expression sets up the entry to the REPEAT path for a second processor. The conditional expression and the REPEAT path are then executed simultaneously. Executing the symbol "REPEAT" forces the second processor to enter the REPEAT state. The environment changes made by a REPEAT state processor do not affect the main storage and are only kept in the local storage of the processor.

The symbol "REPEATEND" is added to the statement. Execution of "REPEATEND" will have no effect on a processor which is in the normal mode of operation. Once a REPEAT state processor executes the "REPEATEND", it will halt its execution in the WAIT state.

When the first processor executes the "UNTIL," it interrupts the second processor. If the result of the conditional expression is TRUE, the second processor becomes free immediately. The first processor then follows the UNTIL pointer to execute the next statement.

If the result is FALSE, the environment changes stored in the local storage of the second processor will be copied into the main storage, and the second processor becomes free. The first processor then resumes its execution from where the second processor was interrupted.

III. Architecture

The architecture of a high-level language computer which can execute the internal language as described in Section II is shown in Figure 3. It consists of PES Memory, Main Memory, Partial Result Storage, a Scanner, an I/O Processor, and a number of PES Access Processors, Entry Point Registers, Syntactic and Semantic Processors, Local Storage, and Arithmetic Processors. The various kinds of processors are operating simultaneously in a pipelined manner, and the organization is similar to the one proposed by Haynes [1] except that multiple identical processors are also used for parallel computations. Since we are interested only in the parallel execution aspects of the architecture, other features which are the same as Haynes [1] will not be duplicated here.

PES Access Processors

The PES Memory stores the internal representation of the source programs. During the translation phase, the PES Access Processor receives program tokens in the internal form from the associated Syntactic and Semantic Processor, assembles and stores them into the PES Memory. During the execution phase, each PES Access Processor reads the program from the PES Memory, separates and delivers the symbols to the associated Syntactic and Semantic Processor that it is attached to. A free PES Access Processor will start executing a (parallel) execution string by using a non-empty value from one of the Entry Point Registers as a starting address in the PES Memory for execution. After that the Entry Point Register is cleared.

The PES Access Processor can continue reading from the PES Memory until either its buffers are full or it has read a semicolon, which indicates the end of a simple statement in a concurrent statement.

Parallel Pointers and Jump Pointers are executed by PES Access Processors. When a PES Access Processor reads a Parallel Pointer, it puts the pointer value and its processor identification into one of the Entry Point Registers and continues its processing. When a PES Access Processor reads a Jump Pointer, it simply alters its program counter and reads the program from the new location.

Syntactic and Semantic Processors

Each PES Access Processor is attached to a Syntactic and Semantic Processor. The PES Access Processor and its associated Syntactic and Semantic Processor are operating concurrently and asynchronously. The communication between them is carried out by the buffers in the PES Access Processor and a counting semaphore. During translation, the Syntactic and Semantic Processor receives program tokens from the Scanner, performs syntax analysis, translates the program into the internal language, and delivers the resulting program to the PES Access Processor. During the execution phase, the Syntactic and Semantic Processor executes various types of operators sent by its PES Access Processor, such as IF, THEN, ELSE, BEGIN, WHILE, REPEAT, etc. It also sends commands to its Arithmetic Processor and the I/O Processor. A Syntactic and Semantic Processor can also alter the program counter of its PES Access Processor when it executes a "GOTO", "WHILE", or "UNTIL." Each Syntactic and Semantic Processor has its own local memory to temporarily store the environment changes during try-ahead processing.

The Syntactic and Semantic Processors are interconnected to each other so that when a try-ahead path is taken by a Syntactic and Semantic Processor, it can send its processor identification to the processor which is executing the conditional expression. After the conditional expression is evaluated, the latter processor will interrupt the former and take the appropriate actions as described in Section II.

Arithmetic Processors

An Arithmetic Processor is connected to each of the Syntactic and Semantic Processors. When a Syntactic and Semantic Processor receives an operand from its PES Access Processor, it saves the type and value of the operand into its operand registers. When it receives an arithmetic operator, it directs its Arithmetic Processor to perform the operation on the operands stored in its operand registers. The Arithmetic Processor will check the types of the operands, and perform all type conversions if needed. The results of an arithmetic operation are stored into the operand registers of the Syntactic and Semantic Processor which has sent the operator. Our scheme used here will not require any stack for arithmetic expression executions, and, at any time, no more than two operands will be in the operand registers of a Syntactic and Semantic Processor. A stack is used in the main storage only to allocate space when a block or procedure is entered.

Partial Result Storage

The Partial Result Storage is to temporarily store the partial results obtained during the execution of an expression. Each location in the Partial Result Storage has a tag associated with it to indicate whether it is empty or full. All tags are cleared initially to indicate "empty." When a Syntactic and Semantic Processor receives a partial result operand, i.e., an operand of the form #i, from its PES Access Processor, it will check the tag of location i in the Partial Result Storage. If it indicates "empty", the Syntactic and Semantic Processor will save the contents of its operand registers into location i of the Partial Result Storage and set the tag to indicate "full". The Syntactic and Semantic Processor then becomes free. If the tag indicates "full," the Syntactic and Semantic Processor will reset the tag to indicate "empty", read the contents of location i into its operand registers, and use them as the operand for the next operation.

IV. Conclusions

In this paper we have presented an internal language for a high-level language computer, in which arithmetic expressions and concurrent statements are expressed as parallel executable strings. Try-ahead operations are performed for IF statements, WHILE statements, and REPEAT statements. For an IF statement, both the THEN path and the ELSE path are tried simultaneously, while the conditional expression is being executed. The wrong path is later discarded, and the right path activated. For WHILE statements and REPEAT statements, only the repetitive path is tried ahead, since it is the one more likely to be correct. The resulting system can increase its processing speed over other designs through distributed processing of various tasks by multiple independent processors, through parallel execution of arithmetic expressions and concurrent statements, and through try-ahead processing of the statements involving conditions.

V. References

[1] Haynes, L.S., "The Architecture of An Algol 60 Computer Implementation with Distributed Processors," Proceedings of the 4th Annual Symposium on Computer Architecture, pp.95-104, March 1977.

[2] Chu, Y., "A LSI Modular Direct Execution Computer Organization," Computer, Vol.11, No.7, pp.69-76, July 1978.

[3] Chu, Y., "Concepts of High-Level Language Computer Architecture," in High-level Language Computer Architecture, (Y. Chu, ed.), pp.1-14, Academic Press, New York, 1975.

[4] Haynes, L.S., "Structure of A Polish String Language for An ALGOL 60 Language Processor," Proceedings of ACM-IEEE Symposium on High-Level Language Computer Architecture, pp.131-140, Nov. 1973.

[5] Wang, P.S. and Liu, M.T., "Parallel Processing of High-level Language Programs," Proc. of the 1979 International Conf. on Parallel Processing, pp.17-25, Aug. 1979.

[6] Wang, P.S. and Liu, M.T., "A Multi-microprocessor System for Parallel Computations," Proceedings of the Second Symposium on Small Systems, pp.59-68, October 1979.

[7] Dijkstra, E.W., "Cooperating Sequential Processes," in Programming Languages, (F. Genuys, ed.), pp.43-112, Academic Press, New York, 1968.

[8] Laliotis, T.A., "Implementation Aspects of the SYMBOL Hardware Compiler," Proceedings of the First Annual Symposium on Computer Architecture, pp.111-115, December 1973.

Appendix

A Translation Algorithm

The algorithm is to translate an arithmetic expression into the internal language form described in Section 2.1. During the translation process, two stacks will be used: OPR-STK and OPN-STK, for storing operators and operands, respectively. Dollar signs ($) will also be used in OPN-STK. LC is the Location Counter which contains the address of the location for storing the next output. Two variables are used: TEMP-COUNTER is for the number of temporary storage locations used, and PES-BEGIN is the starting address of the string currently being generated. An array TEMP-POINTER (TP) is used in the algorithm. TP(i) stores the address of the first of the two #i's in the output, so that when the second #i is generated, a Jump Pointer to the second #i can be generated at the location following the first #i.

The algorithm is similar to that of translating an expression into a reverse Polish string, except that operands are not written out immediately and its operator output procedure is more complicated. A hardware translator can be easily implemented in the Syntactic and Semantic Processor [8]. Figure 4 is the flowchart of the algorithm.

Main Procedure

1. Clear TP array. Initialize TEMP-COUNTER <- 0; PES-BEGIN <- LC.
2. S <- next input symbol.
3. If S is a '(', then push OPR-STK('(') and go to 2,
   else if S is a variable, then push OPN-STK(S),
   else ERROR.
4. S <- next input symbol.
5. WHILE Priority(OPR-TOP) $\geq$ Priority(S) DO POP-OPR-STK.
6. If S is a ')' and OPRTOP='(', then pop OPR-STK and go to 4.
   If S is a ')' and OPRTOP is not '(', ERROR.
7. If S is an arithmetic operator, then push OPR-STK(S) and go to 2.
8. If S is 'end-of-expression' and OPR-TOP is not a '(',
   then DONE else ERROR.

Procedure POP-OPR-STK

Case 1 The OPR being popped is a unary operator:

Case 1.1 OPN-STK(TOP) is a variable:
1. FINISH-PREVIOUS-PES.
2. Pop OPN-STK, and output it.
3. Output OPR.
4. Push $(TEMP-COUNTER + 1) onto OPN-STK.

Case 1.2 OPN-STK(TOP) is a $k:
1. Output OPR.

Case 2 The OPR being popped is a binary operator. Depending upon the top two elements on OPN-STK, there are three cases:

Case 2.1 Both of the two elements are variables:
1. FINISH-PREVIOUS-PES.
2. Output the top two elements from OPN-STK.
3. Replace the top two elements on OPN-STK by $(TEMP-COUNTER + 1).
4. Output OPR.

Case 2.2 One element is a variable, and the other is a $i:
1. Output the variable.
2. If OPR is non-commutative and OPN-STK(TOP) is $i, then output OPR', else output OPR.
3. Replace the top two elements on OPN-STK by $i.

Case 2.3 Both of the two elements are $'s. Let OPN-STK(TOP-1) be $k, and let OPN-STK(TOP) be $j:
1. Output #k.
2. If OPR is non-commutative, then output OPR', else output OPR.
3. Output OPR to the location pointed to by TP(K).
4. Output a Jump Pointer with the content of LC to the location pointed to by TP(K)+1.
5. Replace the top two elements on OPN-STK by $j.

Procedure FINISH-PREVIOUS-PES

1. TEMP-COUNTER <- TEMP-COUNTER + 1.
2. Output #(TEMP-COUNTER).
3. TL(TEMP-COUNTER) <- LC; increment LC by 2.
4. Output a Parallel Pointer with the content of LC to the location addressed by PES-BEGIN.
5. PES-BEGIN <- LC.

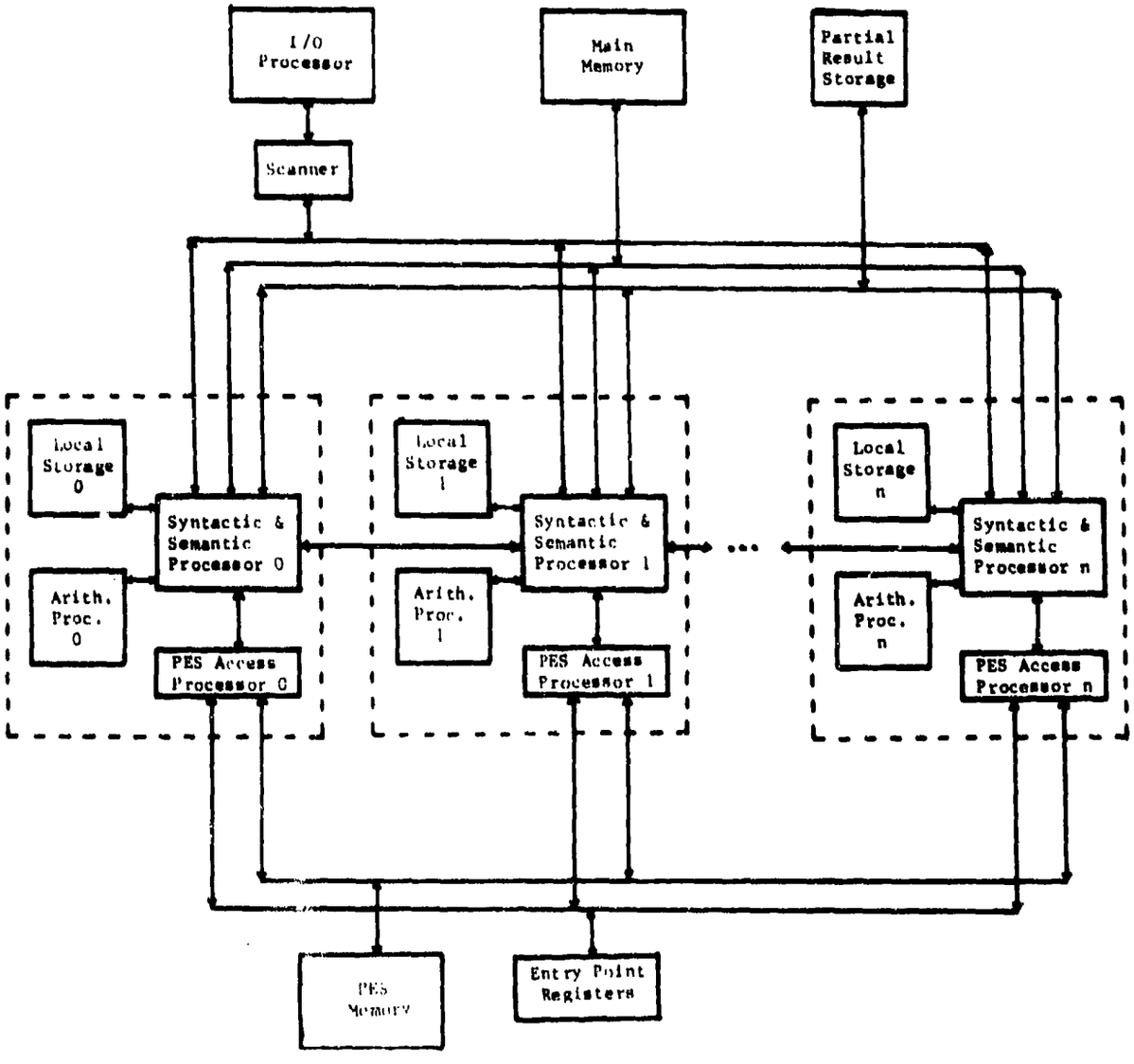


Figure 3 Machine Organization

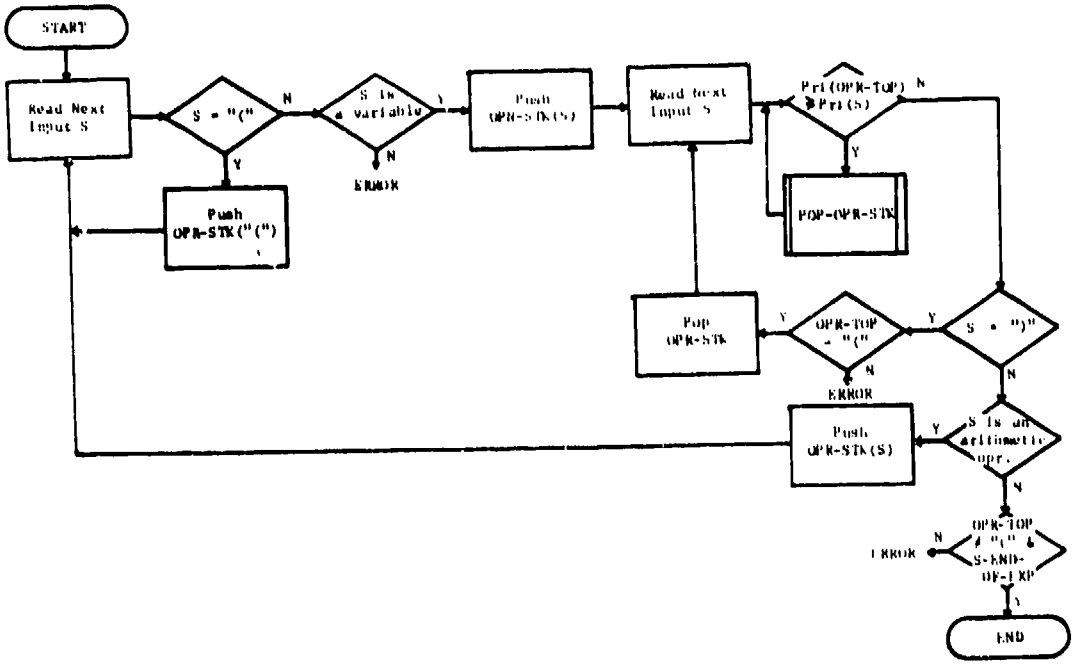


Figure 4a Main Procedure of the Translation Algorithm

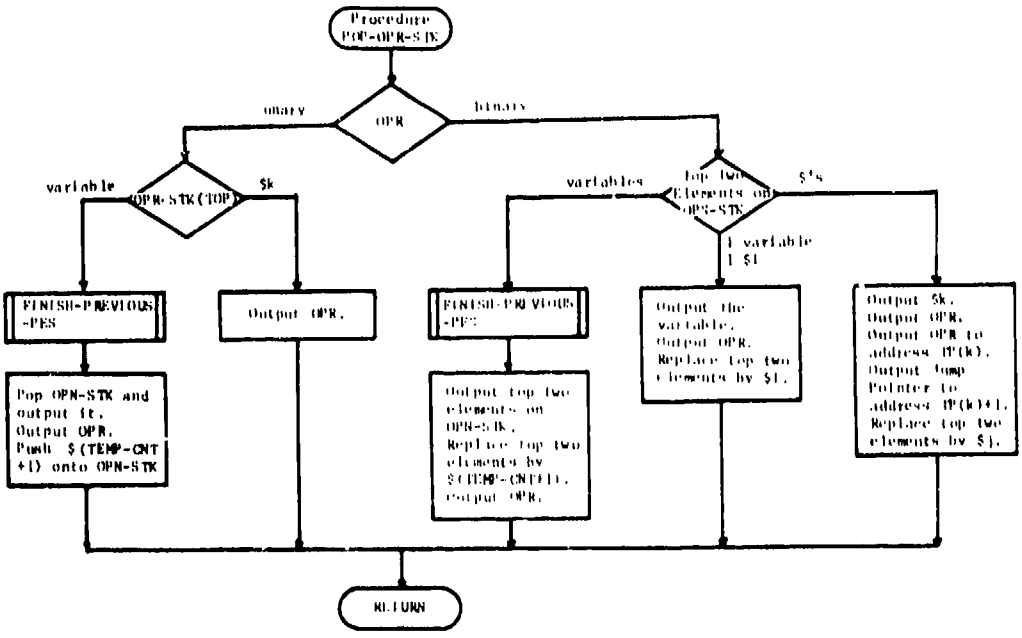


Figure 4b Procedure POP-OPR-STK

DIRECT-EXECUTION HIGH-LEVEL LANGUAGE FORTRAN COMPUTER

Yoory Nain Chen, Kuo Liang Chen, Ke Cheng Huang

University of Science and Technology of China
Hefei Anhui People's Republic of China

Abstract

This paper presents a conceptual design of the direct-execution Fortran Computer. First some modifications are introduced into the language Fortran to insure simpler execution and better performance. Next follows a brief discussion of the architecture of this computer and then, in more detail, of the direct-execution process of some typical Fortran statements which may furnish an outline of the work of this computer. Finally, some comments are made on the possible development of the direct-execution high-level language computer.

I. Introduction

With the rapid advance of the science and technology of computers and electronics, the cost of the hardware becomes cheaper and that of the software becomes more expensive day by day. This makes it both possible and necessary, to design the direct-execution high-level language computer. Since the language Fortran is the most widely used high-level language, the research and design of the direct-execution Fortran computer may not be ill-advised.

This paper gives a conceptual design of the direct-execution Fortran computer. It uses the ANSI basic Fortran as the fundamental language, but in order to insure simpler execution and better performance, some modifications are introduced into this language as follows.

(1) Main program proceeded by the keyword "Master" should be put at the end. For interaction between man and machine inputting is carried out token by token; the main program should be put at the front of the whole program. But at that time the labels of each program unit should be limited to a certain number.

(2) The types of all the variables and arrays should be declared explicitly, especially the dummy arguments of the statement function.

(3) In order to distinguish between the non-executable and executable statements, a keyword "MACRO" is placed at the beginning of each statement function. The dummy arguments of statement function are localized in the program unit of this statement function.

(4) The EQUIVALENCE statement is deleted.

The architecture of the direct-execution Fortran computer is described briefly in section II of this paper. The direct-execution procedures of some typical statements of the language Fortran are discussed in section III. We believe that may furnish an outline of the work of this direct-execution Fortran computer. Finally some comments are made on the possible development of the direct-execution high-level language computer in section IV.

II. Architecture

The direct-execution high-level language computer should execute the program written in this language directly according to its lexicon, syntax and semantics without using the traditional and complicate multilayer software. (such as compilers, assemblers, loaders, etc). Thus, its architecture should reflect the structures of Lexicon, Control and Data of this high-level language, so that the program written in this language may be treated more efficiently.

The computer architecture diagram proposed is shown in Fig. 1. It consists of a Program Memory PM (to store the user's program), a Data Memory DM (to store the relevant data) and four processors (Input/Output Processor I/O P, Lexical Processor LP, Control Processor CP and Data Processor DP). Among the processors there are also the control bus, the address bus, the data bus and some registers to store information

temporarily. These processors may be microprocessors or built up with LSI chips. They may operate parallelly and synchronously with each other in order to increase the processing speed.

The user's program may be input into the PM either all at once, or token by token, executing and storing simultaneously to allow interaction between man and machine. After treatment by I/O P the user's program is input into the PM in a definte form: namely with a terminal character at the end of each statement and two tagging characters one at the beginning of each Program unit and the other at the end of the whole program. These tagging characters are called unit heads and program end characters respectively, they are in the first position of the label region and are different from any ordinary characters used by Fortran. The codes stored in the PM may be either ASCII or compressed internal codes.

LP is used for lexical analysis. It includes the SAM (Scanner Associative Memory which stores legal characters, etc). There are two working modes for LP controlled by CP: scanning and executing. In the scanning mode, LP checks the characters sent from PM whether there is a terminal character or not, so as to find out the label region (since the label region is just next to the terminal character.) After LP finds out the label, the unit head tag and the character "D" at the first position of the statement, LP is transfered to the executing mode. In the executing mode, it checks the legality of characters sent from PM, Spells them into tokens and sends them to CP and/or DP. If the tokens are a string of numbers, set [illegible] register to "1", make some conversion and put the converted codes into the VALUE register and then send them out.

CP consists of the CAMU (Unit Head Control Associative Memory), the CAML (Label Control Associative Memory), the CAMR (Reserved Word Control Associative Memory), the R Stack (Return Stack), the DO Stack, the CALL Stack and the MDLREG (Mode of DP and LP Register). CP is the control center of this computer. When the main program is executed or the subprogram is called it sets DP into the operating mode by means of the register MDLREG. Otherwise it may set DP into the syntax mode. The working modes of LP are also set by means of the register MDLREG. R Stack is used for reserving the return position. DO Stack is used for reserving the DO statement information and CALL Stack for reserving relevant information of local quantities when a call subprogram is executed. When LP outputs a unit head or a label, CP should fill the entries of CAMU and CAML respectively for later use in some control statements concerned.

DP consists of DAM (Data Associative Memory), some stacks (EXP Stack, V Stack, L Stack and P Stack) and register FDMPT. In the syntax mode for non-executable statements (declaration part) it fills the corresponding entries of DAM for the variables and arrays but does not allocate any cells in DM. (except COMMON Statements). For executable statements no treatment should be necessary; it's a matter of starting the LP by CP to continue the scanning. Now as DP is in the operating mode, it not only fills the corresponding entries of DAM, but also allocates cells in DM for them. Then it calculates the values of executable statements and assigns values to them. The register FDMPT points to the first usable location of the free space in DM. (After returning of the called subprogram the space in DM allocated to it should be released for other uses.) EXP Stack stores operators. V Stack stores the values of operands. L Stack stores the logical operands.

Besides, DM is the Data Memory; data stored in it are tagged to indicate the type of data. Scanner pointer SP is a pointer which points to the location of the character being treated in PM. The RESULT register stores the operating results to control the DO and IF statements.

III. Direct-execution of some statements

Before executing we assume that the user's program is stored in PM already. The pointer SP points to the first character of the program in PM and LP is in the scanning mode.

1. Treatment of unit head statements

When LP scans the unit head tag, it is changed to the executing mode, spelling the characters into tokens to be output. CP receives and analyzes the tokens to determine the type, class and name of this program unit. Then it fills these items into the corresponding fields of global CAMU as shown in Fig. 2, where DAMPT points to the first location of the local quantities in DAM, PMPT points to the first character location of the first executable statement of this unit in PM, LPT points to the location of the first label of this unit in CAML. These pointers should be filled before the unit is called.

If this unit is a function subprogram, DP should be activated to fill its name into the DAM of this unit as shown in Fig. 3. Then the location of the first entry in DAM of this unit should be put into the field DAMPT in CAMU. Finally, there should be left a blank between the two neigboring units in DAM, CAML etc, to indicate the end of the units.

For the subprogram with dummy arguments, after CP recognizes a dummy argument, it activates DP to fill the entries of this block successively and put a dummy symbol in the field Dummy. When it encounters the character ")" it fills the number of dummy arguments in the field SIZE.

2. Treatment of declaration statements

When CP encounters the names of variables or arrays of the non-executable statements, it puts them into the DSR (Data Search Register) and then activates DP to find out whether there are such names in DAM or not. If there are, DP fills the corresponding fields. If there aren't it allocates new entries.

The field STRUCTURE indicates that the structure of the name is a variable, an array or a function. The field TYPE indicates that its type is a real or an integer; the field COMMON indicates whether it is allocated in the COMMON region or not; and the field SIZE indicates the volume of array or the number of the dummy arguments. Besides, the field PT1 points to the location of the variable (or the location of the first component of the array) stored in the DM. In our scheme, for all the quantities of non-executable statements except those specified by the COMMON statements, we do not allocate any cells in DM, that is to say, we do not fill PT1 until this unit is called by another program unit. Of course, for these quantities in the main program cells are allocated. The pointer PT2 stores the information for calculating the location of the components of an array, so it is filled for arrays only.

3. Treatment of the statement function

The treatment of the statement function is to fill its name and dummy arguments together with their types into the DAM but not to allocate any cells in the DM. When DP encounters the token"=", the content of SP should be filled in the field PT3 in the DAM and then CP sets LP to scan until the terminal symbol of this statement is encountered.

4. Treatment of the Definitional Label

Before encountering the first executable stat statement of the main program, DP is in the syntax mode, i.e. it only treats the non-exectable statements as discussed above while for the executable statements it treats the definitional label only.

The treatment of the definitional label is to fill the label in the entry of the CAML of its unit according to the sequence of its appearing in the program as shown in Fig.4, where PMPT (Program Memory Pointer) points to the location of the first character of the statement of this label in PM. DL indicates the number of nesting of DO Loops and is used for preventing the program to transfer into inner layer of the loops.

For the executable statements of the subprogram, LP is set in the scanning mode to scan the label region and the first character of the statement. Now if the label is encountered, CP fills one entry of CAML and "0" in its DL field to indicate that the label is not in any loop body.

If the first character of the statement is not the alphabet "D", then the LP should scan continuously. If it is, the LP should output a token. When CP does not encounter the keyword "DO", it sets LP to scanning mode again; if it does encounter "DO" the following statement should be a DO statement such as:

$$\text{DO } L \quad i = m_1 , m_2 , m_3$$

Then CP pushes the label of the terminal statement L into the field TL of the DO stack (remains the other fields blank) and pushes the return location (i.e.the first character of the loop body) into the R stack as shown in Fig.5.

When a definitional label is encountered CP fills an entry of CAML and "0" into its field DL as well. When the label of the terminal statement L is encountered, besides filling it into the CAML, the DO stack and R stack should be popped, the values of the DL of all labels within this DO loop should be increased by "1".

For multi-nested DO Loops, say, with three nested layers, after SP transfers out of all the DO loops the DL value of the innermost layer is 3, that of the middle layer is 2 and that of the outermost layer is 1. The treatment of definitional labels in the main program will be discussed in the next paragraph.

5. Treatment of DO statements.

When DP is in the operating mode to execute the DO statement "DO L i = m_1, m_2, m_3", if the terminal statement label L is found in the CAML, the PMPT of L is pushed into the field TL of the DO stack, m_1 is assigned to i, and the locations of i, m_2 and m_3 are pushed into the fields CV (Control variable), FP (Final Parameter) and IP (Incremental Parameter) of DO stack respectively. The return location is pushed into R stack (Return Stack) also, as shown in Fig.5. Labels which are found in the CAML with addresses both less than or equal to that of the terminal label L and greater than or equal to that of the R stack are within the Loop body. Then the values of the field DL of all the labels within the loop body should be decreased by "1". The values of DL of this layer now equals to "0", which indicates that these labels are in the same layer, so that they may be transfered. When the nested DO statement is encountered CP goes through all the procedure as discussed above. Having executed the terminal statement of the DO loop. CP activates DP to calculate i = i + m_3 and RESULT = i - m_2 . If RESULT≠0, then the top items of R Stack should be copied into SP to make the loop execution again. If the RESULT>0, the loop execution is completed, the values of the field DL of all the labels within the loop body should be increased by "1", It indicates that these labels within this loop should not be transfered. Then the DO stack and R stack should be popped.

If L is not found in CAML the statements in the loop body should be executed. When the definitional labels are encountered, CP allocates the entries of CAML to them and fills the fields DL with "0". Having executed the terminal statement of the DO loop, treat them as discussed above.

6. Treatment of GOTO and IF statements

If the GOTO L statement is not in any DO Loop, and the label L is found in the corresponding region of CAML, CP checks the value of the field DL; if it is "0", the program may transfer to the label L; otherwise, an error has occurred. If the label L is not found in the corresponding region of CAML, CP sets LP and DP into scanning and syntax modes respectively, scanning the program to find the L. The treatment is similar to paragraph 4.

When the L is found in the program, in order to prevent transfer into the inner layer of DO loop from the out layer, the L should not be transferred immediately (although the value of its fields DL is "0" at that time). The location of L in CAML should be stored in the temorary register TR. LP scans the program continuosly until it returns to the same layer of this GOTO L statement, i.e. the DO loop layer whose fields CV, FP and IP in the DO stack are blank should be scanned out and the values of field DL should be increased. Then the values of the field PMPT and DL in CAMU should be found out by means of the content in TR. If the value of the field DL is "0" then the program transfers to L; otherwise an error has occurred.

For the GOTO L statement lying in a certain nesting DO loop layer, it is necessary to find out L within the current nested layer of CAML. (If it is not found in CAML LP should be set in the scanning mode to scan the program to find the L of this DO loop layer in the program as discussed above). If L is found and its DL value is "0", the program should be transferred to L. If L is not found, the values of the field DL of all the labels within this loop layer should be increased by "1". CP pops the top of DO stack and R stack; goes on to find the label in the outer layer (in the CAML or in the program). The above process is repeated again and again until the L is found and the program is transferred to it.

The execution of IF statement IF (e) L_1, L_2, L_3 is similar to the GOTO statement. When CP recognizes the keyword "IF" it activates DP to calculate the expression and puts its logical result (less than, equal to or greater than zero) into the RESULT register. Then according to this result CP puts the value of the corresponding PMPT of L_1, L_2 or L_3 into SP to perform the transfer.

7. Treatment of the call of statement function subroutine and function subprogram

In the case of calling a function subprogram or a statement function, it is necessary to find the name of the function in the region of the current operating program unit of the DAM. If this name is found it is a statement function; otherwise, it may be a function subprogram. For a function subprogram, its name sould be found out in the CAMU.

CP copies the values in the fiels PMPT, DAMPT and LPT of the CAMU into the fields of temporary register. CP allocates a cell in DM for the function name to store values of the function. Then the CP recognizes the actual arguments, say, there are three arguments: A (variable), 3 (constant) and C + D (expression). CP activates DP to find out (by DAM) the location in DM allocated for A. The locations of A and those of temporary cells allocated for constant 3 and the result of expression (C + D) together with the location of the function name should be copied into the fields PT1 of dummy arguments and the function name of the called subprogram in DAM respectively. Then DP pushes the value of the field PT1 of the function name into P stack, as shown in Fig.6.

In our scheme we use call by name. Certainly during the process of substitution some syntax checking (as on whether the numbers of the actual and dummy arguments are equal, whether the types of both arguments are the same, etc.) should be made. When the character ")" has been treated, the return location in DM (the value of SP) should be pushed into R stack, the values of DAMPT and LPT of TR and those of FDMPT pushed into the CALL stack as shown in Fig.5. The value of PMPT of CAMU in the temporary register should be put into the pointer SP to perform the transfer. In executing the executable statements of the function subprogram, DP should allocate cells in DM for local variables which have not been allocated yet, and should modify FDMPT also . Once the RETURN statement is encountered CP puts the value of R stack into SP, pops the CALL stack and R stack and clears all the fields PT1 of the DAM of this subprogram. The calculated result is now automatically available in the cell of the function name.

The call of a statement function is very similiar to that of the function subprogram but the value of PT3 of the statement fouction in the DAM should be put into SP instead of PMPT in CAMU of the function subprogram. At the same time, it is not necessary to alter the CALL Stack.

Since the call of a subroutine statement is preceeded by the keyword "CALL", it is easier to recognize. The treatment of the call subroutine is rather similar to that of a function subprogram.

IV. Conclusion

The language Fortran has been in use for many years in scientific computation and is familiar to most computer users. Since however, quite a lot of trouble is involved in the use of the language Fortran for direct-executing we have to modify it properly in designing the high-level language computer.

Today, the computer hardware and the computer software hold a relation of mutual impetus, mutual penetration and mutual constraint. The development of the computer language and programming has greatly affected computer architecture, as is shown in the improvement from classical computer architecture to high-level language computer architecture. On the other hand, the development of computer architecture also leads to a development of languages, such as the HLM language proposed by Prof. Yaohan Chu.

To sum up, the development of the high-lev level language computer should lead to a close merging of the programming language and the computer architecture; that is, the language and the computer architecture ought to execute the program effectively and the programming also ought to satisfy the requirments of the language and architecture, so as to improve the reliability and practicality as well as the cost-efficiency of the whole system. So in the long run. It is necessary to reconsider and redesign new language from the point of view of programming and computer architecture. Indeed the conception of structure programming and structured language has appeared already, but the languages evolved are not solely dedicated to the high-level computer

Since the said HLM language has not won wide acceptance yet, we think it necessary to design some new computer architecture for the current language such as Fortran, Cobol etc. This is our motivation in writing this paper.

V. Acknowledgement

We are grateful to Prof. Yaohan Chu of University of Maryland U.S.A. for his kind help and active support.

We are also grateful to those of our University --- University of Science and Technology of China --- who have been kind enough to provide various facilities for making this paper possible.

References

(1) Yaohan Chu "Direct-Execution in a High-Level Computer Architecture", Proceedings of the ACM Annual Conference, Washington D.C., December, 1978.
(2) Yaohan Chu "An LSI Modular Direct-Execution Computer Organization", COMPUTER, IEEE Computer Society, July, 1978, pp. 69-76.
(3) Yaohan Chu and Cannon, E.R." A Programming Language for High-Level Architecture", Proceedings of the National Computer Conference, New York City, June, 1979, pp. 657-665.
(4) Yaohan Chu and Cannon, E.R. "Interactive High-Level Language Direct-Execution Microprocessor Systems", IEEE Transactions on Software Engineering, June, 1976. pp. 126-134.
(5) ISO Recommendation R 1539 PROGRAMMING LANGUAGE FORTRAN.
(6) ANSI, Basic FORTRAN, X3.10-1966.
(7) David Gries, Compiler Construction for Digital Computer, John Wiley & Sons, Inc. New York 1971.
(8) [illegible] FORTRAN [illegible] 1979.12.

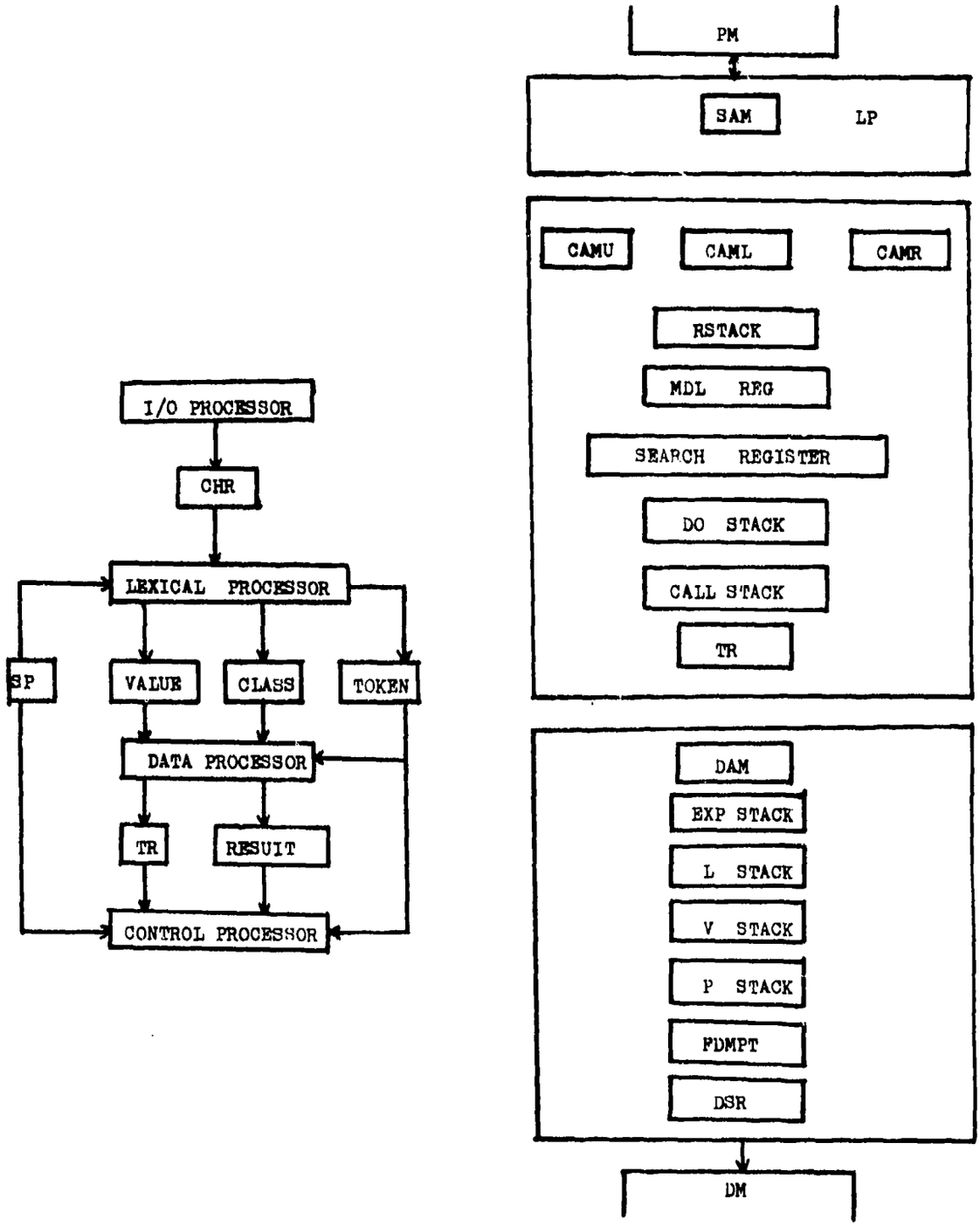


Fig.1 Organization of the Direct-execution FORTRAN Computer.

C A M U					
NAME	TYPE	CLASS	DAMPT	PMPT	LPT
f	REAL	FUNCTION			
⋮	⋮	⋮	⋮	⋮	⋮

Fig.2 The Unit Head of the Control Associative Memory

D A M								
NAME	STRUCTURE	TYPE	SIZE	COMMON	DUMMY	PT1	PT2	PT3
f	FUNCTION	REAL	2		DUMMY			
X					DUMMY			
y					DUMMY			
B	ARRAY	INTEGER	50	COMMON				
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Fig.3 A Data Associative Memory

C A M L		
LABEL	P M P T	D L
⋮	⋮	⋮

Fig.4 The Label Part of the Control Associative Memory.

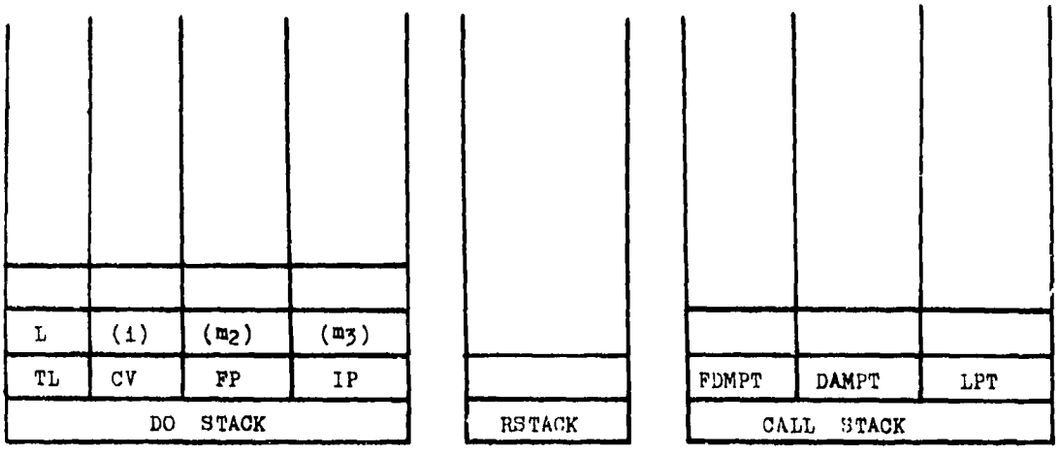


Fig.5 a DO STACK, a Return STACK and a CALL STACK

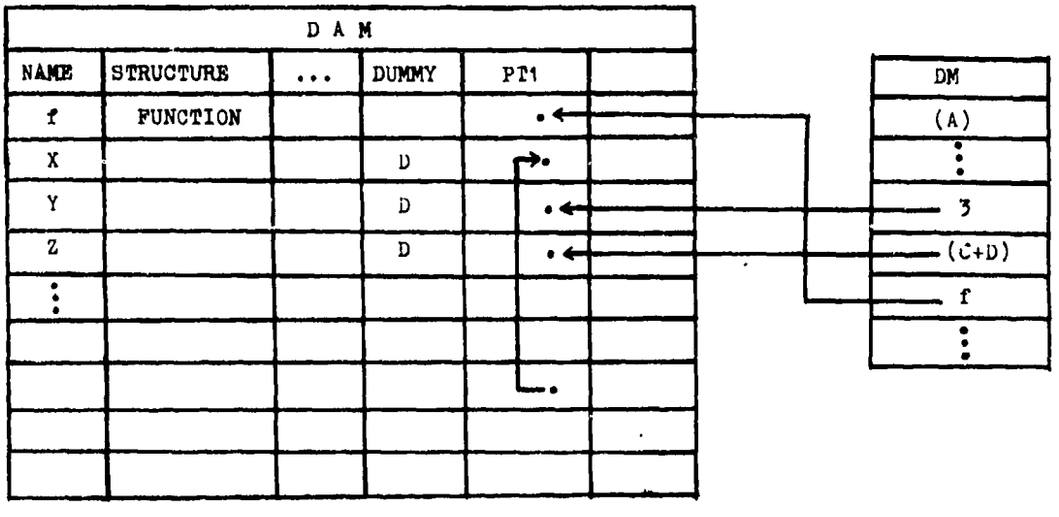


Fig.6 Illustrating the process of the substitution of the Actual argnments for the Dummy Illustrating arguments.

A JOVIAL DIRECT EXECUTION COMPUTER

Yaohan Chu

Department of Computer Science
University of Maryland
College Park, MD 20742
301-454-4245

Abstract

This paper reports a JOVIAL direct-execution machine which accepts a subset of the JOVIAL J73 language. It describes the J73 subset. It also describes the organization of the JOVIAL direct-execution computer which reflects the language constructs of the J73. There are 3 processors, 3 associative memories, a program memory, a data memory and 10 interfacing registers. The memory/register/stack structures and direct-execution algorithms of the processor are described.

1. Direct Execution Computer

Direct-execution refers to the operating mode of a high-level architecture. This operating mode directly accepts and executes a high-level language program without the need of multiple layers of conventional software. As a result, there is no compiler, no assembler, and no linkage editor. The high-level programming language is the machine language that the bare hardware recognizes. A direct execution computer is capable of operating in the direct-execution mode.

The direct-execution computer [9] is structured with a direct execution cycle; this is shown in Fig. 1. A high-order language Program is stored in the program memory. The lexical processor fetches the next token from the program memory and delivers the token to the language processor; the language processor executes the token accordingly. This cycle continues until the program ends.

The direct-execution computer [1,7,8] is organized to reflect the constructs of a high-level programming language. The organization is shown in Fig. 2, where there are: a program memory PM, a data memory DM, three associative memories (SAM, CAM, and DAM), and three processors (lexical processor LP, data processor DP, and control processor CP). The program memory stores the source program. The data memory stores the data values. The associative memories store descriptors which represent the data and control information in the source program. After initialization, the control processor fetches the next token from the lexical processor which has access to the program memory. It then either executes the token or activates the data processor to execute it. This process of direct-execution token-by-token continues until the source program reaches the end.

This paper describes a JOVIAL direct-execution computer, which makes use of the above-mentioned direct-execution organization.

2. A JOVIAL Machine

The JOVIAL computer in this paper is designed for a subset of the revised MIL-STD-1589A (USAF) definition of the upgraded J73 JOVIAL programming language dated MARCH 15, 1979 [11].

2.1 A J73 Subset

The J73 is a dialect and an outgrowth of the ALGOL 60 programming language [10]. As a result, the J73 retains a great deal of the ALGOL 60 language. It is a complex compiler-oriented language. A subset of J73 is chosen. There are 46 syntactical statements. The syntactical constructs are outlined below.

(a) Program Structure

The subset allows the complete program to have a main-program module and zero or more procedure modules. The main program module must be the first module. Its construct is shown below.

START PROGRAM < name >; < program body > TERM

The construct of the program body is the same as the procedure body except the former permits directives.

(b) Declarations

There are four types of declarations: item, table, external, and define. The first two declare the data elements, while the third declares a procedure module. The last is a macro for text substitution. The declarations may be enclosed by a pair of 'BEGIN' and 'END' to become a block declaration.

(c) Procedures and Functions

There is both procedure declaration and procedure definition. The procedure declaration is for use in the external declaration. When a procedure definition is enclosed by a pair of reserved words 'START' and 'TERM', it becomes a procedure module. It permits formal parameters. There is one function 'FLOAT (<number>)' which converts an integer into a floating number.

(d) Statements

A statement can be simple or compound. There are four types of simple statements: assignment, loop (or FOR-statement), IF, and procedure-call. Statements may be enclosed by a pair of 'BEGIN' and 'END' to become a compound statement.

(e) Formulas

There are three types: integer, floating, and boolean. An integer formula represents an integer, while a floating formula represents a floating-point number. There are four operators ('+', '-', '*', and '/') for both integer and floating-point operations. A boolean formula represents a value of true or false. There are six relational operators.

(f) Data References

There are two types of data references: variable and function-calls. A variable can be an item or a table. As mentioned, there is only one intrinsic function.

(g) Lexical Elements

There are 56 characters which are grouped into 26 letters, 10 digits, and 20 marks. There are 4 basic lexical elements: token, comment, define, and trace. A token can be a name, a number, a floating-literal, or an operator. There are 34 operators which include 15 reserved words. A comment is a string of characters enclosed by a pair of quotation marks. A define has as its body also a string of characters enclosed by a pair of quotation marks. Only one directive is permitted; this directive has as its body a series of names separated by commas.

2.2 A Sample Program

A J73 sample program is shown in Fig. 3. The line numbers are not a part of the program; they are used for references. The numbers are the same for the two parts of the program; they are distinguished by being referred to as upper lines and lower lines.

The upper lines 00000 to 02000 indicate the start and the termination of the complete program. The complete program consists of a main program module and a procedure module. The main program module consists of program name TSIJOV (upper line 00100), program body (lines 00200 to 01900). The program body has an external declaration (upper line 00300) of Proc TRIG which includes a block declaration of items ANG, SANG, and CANG (upper lines 00400 to 00800), three declarations of tables DEG, SSIN, and CCOS (upper lines 00900 to 01100), a declaration of item II (upper line 01200), a directive of TRACE (upper line 01300), and a FOR statement (upper lines 01400 to 01900). In this FOR statement, there is a call of procedure TRIG (line 01700).

The procedure module begins and terminates at the lower lines 00000 and 02200, respectively. It has a procedure heading (lower line 00100) and a procedure body (lower lines 00200 to 02100). The procedure body has a define declaration (lower line 00200), declarations of six items (lower lines 00300 to 00800), a comment (lower lines 00900 to 01000), three assignment statements (lower lines 01300 to 01500), and a FOR statement. The controlled statement of the FOR statement is a compound statement which consists of an assignment statement and an IF statement.

2.3 Computer Organization

The organization of JOVIAL direct-execution computer [13] is developed from the direct-execution computer organization in Fig. 2. It is shown in the diagram in Fig. 4 where there are the following computer elements:

(a) 3 processors: LP, CP, and DP,
(b) 3 associative memories: SAM, CAM, and DAM
(c) 2 random access memories: PM and DM
(d) 2 tables in ROM,
(e) 10 interfacing registers, and
(f) main bus

The memory/register/stack structures of the 3 processors together with the interfacing registers are shown in Fig. 5. The functions of these 10 interfacing registers are described below.

(a) Register SPTR points to a character in Program Memory. It is of special importance when marking the location and other unique pointers of control statements and procedure modules, the bodies of define declarations, and the return position from procedure and define calls.

Except for the very first call for a token, SPTR is set at the first character of the next token to be formed when a token is requested. After the token has been formed by the Lexical Processor, SPTR is advanced, if necessary, to point to the first character of the next token.

(b) Register TOKEN holds the last token formed by the LP. This register is referenced by nearly every sequence, since the tokens define the program.

(c) Register TYPE stores the type of a name. A name may be a reserved word ('R'), pseudo-function call ('FLOAT'), trace-directive name ('DIR') or identifier ('N').

(d) Register BLOCK stores the top entry of the Control Processor's BSTACK. It identifies the module which the program is currently executing so scope checks can be made on declared names.

(e) Register BACK-PTR saves the position of the first character of the current token.

(f) Register D-LEV stores the top entry of the Lexical Processor's RETURN stack. (An empty stack gives D-LEV a value of zero.) This value identifies a specific define call or that there is no active define call (it can be considered a 'define-activation-level'). The register's purpose is to protect the CP from creating CAM entries for control statements whose Program Memory Pointers are in different define bodies.

(g) Register DEF-DCL is a flag which identifies whether a define declaration is permitted or not. Define declarations follow all the rules associated with other declarations.

(h) Register DEF-CALL is a flag which identifies whether or not a define call is permitted. A define call is not allowed when the next token expected is a module name or declaration name.

(i) Register PROC-NAME saves the name of a procedure when the procedure is called. The register is used to match a procedure module name on the first call of the procedure, and as a switch to determine if the procedure heading and declaration list must be processed (first time) or skipped over (second time).

(j) Register RESULT holds the information about the value, type and structure of formulas and variables. It is used by the Data Processor to calculate and pass values to the Control Processor.

3. Control Processor

The control processor CP directly executes control constructs such as conditional branch, procedure call, nesting, and looping of the J73 subset. It also creates and stores the control descriptors in the control associative memory CAM. These control descriptors can expedite the repeated execution of statements in a program loop without the need for repeated syntactical processing.

A J73 program specifies a sequence of data operations. The sequencing is specified by control statements. The control processor recognizes the control reserved words and then manipulates the pointer in register SPTR (which points to the next character in execution of the source program) of the LP processor to carry out the sequencing.

The structure of the CP consists of one associative memory and 5 stacks as shown in Fig. 5. The functions of these memory and stacks are described below.

(a) Control Associative Memory CAM is to speed up statement execution by saving critical information about control statements and procedure modules. There are three types of CAM entries: if-statement, loop-statement, and procedure-module. The type of entry is stored in the Type field. Information for control statements consists of fields for the location of the statement (for identification), else-part pointer for if-statements, increment formula pointer for loop-statements, and an exit pointer to point to the token following the statement. Procedure CAM entries store the name, location, formal-parameter-list pointer, and body pointer of procedure-modules.

The CAM entries for some control statements composed of define-calls cannot be made. Unless the Program Memory Pointers of a control statement's CAM entry are all on the same 'define-activation level', the proper stack management steps of define-calls and returns may not be followed when SPTR is jumped. When this type of statement is encountered, it will always be treated as a 'first-time', so SPTR is adjusted by repeated calls of sequence NEXT-TOKEN.

(b) Stack BSTACK saves the body pointers of program-body and active procedure modules. Each entry uniquely identifies a module. The top entry of BSTACK is stored in register BLOCK to identify the currently executing module.

In addition, the positions of 'BEGIN's before the first simple-statement of a module body are pushed onto (and then popped off) the stack. This is necessary because both compound-declarations and compound-statements are delimeted by 'BEGIN' and 'END'.

(c) Stack RSTACK saves the return position of procedure calls. The token position following an executed procedure-call-statement is pushed onto the stack. It is restored into register SPTR after the procedure-body is executed.

(d) Stack CTR-STACK's top entry serves as a counter of the 'BEGIN's pushed onto BSTACK and the parameters in a parameter list.

(e) Stack SPTR-STACK has two fields: LOCN holds the location of an active control statement, and DLEV holds the define-activation-level of the location pointer. Before a control statement's PM pointer field is assigned a value, the current activation-level must equal that on the SPTR-STACK. If they are ever different the statement's CAM entry cannot be kept - the location field must be erased. Loop-statements must have their increment and exit pointers on the same level. If they are not, the statement is considered illegal.

The control processor CP directs the control flow. It activates both the data processor DP and the lexical processor LP. processes the following control constructs:

(a) Program structure
(b) Procedure definition
(c) External declaration
(d) Statement
(e) IF statement
(f) FOR statement
(g) Proc-call statement
(h) Declarations

In the following processor design, sequence NEXT-TOKEN, which fetches the next token from the source program, is executed by processor LP as will be described later.

3.1 Program Structure

The program structure consists of those control constructs which form a complete program. These are shown below.

```
1. < complete-program>    ::=< main-program module>
                              [< procedure module>...]
2. < main-program-module>::=START
                            PROGRAM
                            < name> ;
                            < program-body>
                            TERM

3. < program body>        ::=BEGIN < decl-list>
                              [< directive >...]
                              < statement>...
                              END

4. < procedure module >   ::=START
                              < procedure-definition>
                              TERM
```

The above syntax calls for the following hardware sequences.

```
01 COMPLETE-PROGRAM
   02 INIT
   02 MAIN-PROGRAM-MODULE
      03 PROGRAM BODY
   02 CAM-CHECK /*check the variables in CAM*/
   02 NEXT-TOKEN /*processed by LP*/
```

Most of the names of these sequences reflect the terminals or non-terminals of the control syntax. The level numbers indicate the hierarchical relationship of these sequences. These sequences are briefly explained below.

(a) Sequence COMPLETE-PROGRAM. This sequence reflects the syntax that the complete program has one main-program module followed by 0 or more procedure modules.

(b) Sequence INIT. This sequence sets the registers to zero and empties the stack.

(c) Sequence MAIN-PROGRAM-MODULE. This sequence is identified by three reserved words and a semicolon as follows,

START PROGRAM...;... TERM

This sequence identifies the main program module. It calls sequences NAME and PROGRAM-BODY.

(d) Sequence PROGRAM-BODY. The program body is a series of 0 or more declarations followed by one or more statements, enclosed by a BEGIN/END pair. The presence or absence of a declaration has to be determined by the first token of the declaration.

(e) Sequence CAM-CHECK. This sequence searches the CAM for an entry whose name field is the same as the contents of register TOKEN. If it is not found, it returns; otherwise, it is an error.

(f) PROC-MODULE

A procedure module is identified by two reserved words as follows.

START...TERM

However, there is no need for sequence PROC-MODULE since each will be searched and called as a result of a procedure call.

3.2 Procedure Definition

The procedure definition specifies a procedure structure. The syntax is shown below.

```
5. < procedure definition> ::= < procedure heading>;
                               < procedure body>;

6. < procedure-heading>    ::= PROC < name >
                               [< formal parameter
                                        list>]

7. < procedure-body>       ::= BEGIN < decl-list>
                               [< statement>...
                               END
```

The above syntax call for the following hardware sequences

```
03 PROC-DEF
   04 PROC
   04 DEF-HEADING
   04 PROC-BODY
      05 PARA-CHECKS
      05 PARA-POP
```

These sequences are explained below.

(a) Sequence PROC-DEF. This sequence calls sequence PROC-HEADING and then calls sequence PROC-BODY.

(b) Sequence PROC-DEF-HEADING. Sequence PROC-DEF-HEADING checks the syntax of the procedure-heading and sets the parameter pointer and body pointer of the procedure's CAM entry.

(c) Sequence PROC-BODY. A procedure body is similar to a program body, except for two special considerations: the formal parameters must be declared, and the declaration list is skipped over after the first call of the procedure.

(d) Sequence PARA-CHECKS. This sequence checks whether the types and structure of actual and formal parameters agree.

(e) Sequence PARA-POP. This sequence calls DP-sequence PARA-RESTORE to pop each of the procedure's parameters of the parameter stacks in the DP and to return into result of output parameters.

3.3 External Declaration

The external declaration declares an external procedure. The syntax is shown below.

8. < external-declaration> ::= REF < procedure-heading>]
[< declaration>]

The above syntax calls for the following hardware sequences:

01 EXTERNAL-DECL

02 SCAN-DECL

02 PROC-HEADING

03 PROC

03 FP-LIST-CHECK

These sequences are explained below.

(a) Sequence EXTERNAL-DECL. The external declaration is recognized from the reserved word 'REF' which is then followed by a call of sequence PROC-DECL.

(b) Sequence SCAN-DECL. This sequence skips the declarations of the external procedure's parameters.

(c) Sequence PROC-HEADING. This sequence identifies procedure names and their parameter lists.

(d) Sequence PROC. This sequence fetches the proc name and then searches for it in the CAM. If it is not found, it creates a CAM entry for the procedure and inserts the name in the entry.

(e) Sequence FP-LIST. This sequence counts and checks the syntax of a formal parameter list.

3.4 Statement

A statement can be a simple statement or a compound statement. There are 4 types of simple statements: if, for, proc-call, and assignment. The first three statements are executed by the CP. The assignment statement is executed by the TP. Two additional statements, define and comment, are handled by the LP. The syntax is shown below.

9. < statement>::=< simple-statement>
| <compound-statement>

10. < simple-statement>::= <assignment-statement>
! <loop-statement>
! < if-statement>
! < procedure-call-statement>

11. < compound-statement>
::=BEGIN < statement>...END

The above syntax calls for the following hardware sequences:

01 STMT

02 COMPOUND-STMT

02 SIMPLE-STMT

These sequences are explained below.

(a) Statement. Sequence STMT calls sequence SIMPLE-STMT or sequence COMPOUND-STMT by the absence or presence of 'BEGIN' respectively.

(b) Compound Statement. Sequence COMPOUND-STMT calls sequence STMT one or more times.

(c) Simple Statement. Out of the four types of simple statements the IF and LOOP statements can be positively identified by 'IF' and 'FOR', respectively. On the other hand, proc-call and assignments begin with a name. However, the proc-call statement begins with a procedure name which must have been declared and should be found in the CAM. If it is not found, the name is assumed to be the data name of an assignment statement.

3.5 Loop Statement

The loop statement in Machine A is the FOR-statement. It has a control variable, an initial value, and an incremental value. There is additionally a while-clause which sets the condition to terminate the looping. The syntax is shown below.

12. < loop statement> ::=FOR< variable>:< integer formula>
BY <integer formula>
WHILE < boolean-formula>;
< statement>

The above syntax calls for the following hardware sequences.

```
01 LOOP-STMT
   02 SCAN-STMT
   02 FIRST-TIME-LOOP
   02 REPEAT-WHILE
```

The loop statement faces 3 considerations: (a) looping, (b) nesting of loop statements, and (c) first-time problem.

(a) Looping. The looping requires computation of new value of the control variable and evaluation of the boolean formula. If the evaluated result is true, the loop body is executed, and if the loop body is executed for the first time, the EXIT-PTR is inserted in the CAM entry. If the evaluated value is false, the loop's statement is scanned and the EXIT-PTR marked, or execution is directly jumped to EXIT-PTR.

(b) Nesting of Loop Statements. The nesting of loop statements (and if statements) is handled by pushing its CAM entry LOCN fields onto stack SPTR-STACK at the beginning of the sequence and by popping it off at the end.

(c) First-Time Problem. If no CAM entry exists for this statement, one must be created. The location and increment formula position are stored in addition to the EXIT-PTR.

3.6 If Statement

The If statement causes conditional branching. The syntax is shown below.

13. < if-statement>::=IF < boolean-formula>;
< statement>
[ELSE < statement>]

The If statement faces 5 considerations: (a) branching, (b) nesting of If statements, (c) first-time problem, and (d) optional else-clause. These considerations are discussed below.

(a) Branching. The branching requires evaluation of boolean formula. If the evaluated result is true, the Then-clause is executed and the execution continues at the location indicated by the ELSE-PTR if it exists, and otherwise the EXIT-PTR.

(b) Nesting of If statements. The nesting of If statements is handled by pushing the LOCN fields of their CAM entries onto stack SPTR-STACK at the beginning of the sequence and by popping it off at the end.

(c) First-time or Second-time. During the first time, if the boolean formula is true, the then-clause is executed but the else-clause is scanned by sequence SCAN-STMT. If the boolean formula is false, the then-clause is scanned by sequence SCAN-STMT, but the else-clause is executed. During successive times, no scanning is needed since all pointers have been established.

(d) Optional Else-clause. The else flag is available in the CAM entry to indicate whether there is an else-clause. If there is, the else-flag is set and the ELSE-PTR is inserted.

3.7 Procedure Call Statement

The procedure-call statement invokes the execution of a procedure definition. It should be noted that the procedure definition may occur before or after a procedure-call statement. If it is before, the location of the procedure definition can be found from the CAM. If it is after, the program execution has to be suspended and the source program is scanned until the procedure definition is found. The syntax of the procedure call statement is shown below.

14. < procedure-call-statement> ::=< name>
[< actual-parameter-list>];

The above syntax calls for the following sequences.

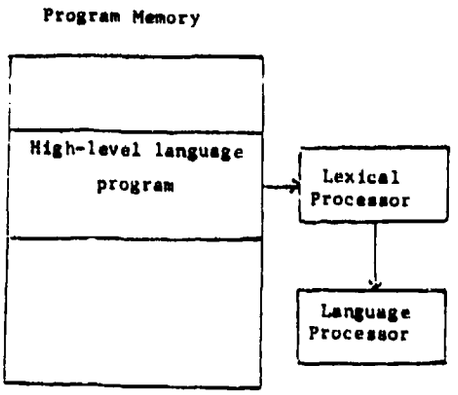
```
01 PROC-CALL-STMT
   02 SCAN-UNTIL-PROC
   02 PROC-DEFINITION
```

The procedure-call statement faces 5 considerations: (a) existence of parameters, (b) nesting of proc calls and returns, (c) first-time problem, (d) ahead or behind a proc definition, and (e) Call-by-value or by-reference. These considerations are discussed below.

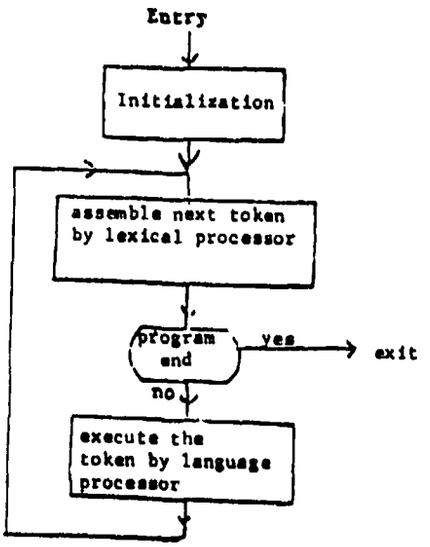
(a) Parameters. The parameters may or may not exist. They can be input or output parameters. Their presence is determined by the parameter count field of the procedure's CAM entry. The DP is then activated to execute sequence ACTUAL-PARApLIST.

(b) Nesting of Proc Calls and Returns. When a proc-call statement is encountered, the return address of the calling procedure is pushed down onto RSTACK. When the execution of a procedure reaches the end, the return address is obtained from the top entry of RSTACK and the entry is then popped off.

(c) First-time Problem. If the PROC-NAME register is not empty, the procedure is called for the first time. During the first time, program execution is now changed into program scanning until the procedure definition is found. This identification is achieved by comparing each procedure name encountered during scanning with that in the NAME field of the top entry of RSTACK. The scanning is done by sequence SCAN-UNTIL-PROC.



(a) Program storage



(b) Program Execution

Fig. 1 Program Storage and Direct Execution of a High-level Language Program

(d) Second-time Problem. The declaration list of the procedure body is not processed on succeeding calls.

(e) Call-by-value or by-reference. The parameter passing in the J73 as follows.

(1) Formal-input parameter: it must be an item, it is bound by value.

(2) Formal-output parameter: if it is an item, it is bound by value-result. If it is a table, it is bound by reference.

(3) Actual-input parameter: it can be an integer or a floating formula.

(4) Actual-output parameter: it must be a variable.

The evaluation and passing of parameters are handled by the DP.

3.8 Declarations

The declaration statements consist of:

```
15. < decl-list > LL=(<declaration>
                      !<define-declaration>
                      ! BEGIN < decl-list>END)...
16. < declaration>::=<item-declaration>
                     !<table-declaration>
                     !<external-declaration>
```

The above syntax calls for the following hardware sequences.

```
01 DECL-LIST
   02 DECL
```

(a) Sequence DECL-LIST processes the declarations of a program-body or procedure-body. Names may not be declared twice in the same module, nor duplicate a procedure-name. A define-call is not permitted when the name of a declaration is the next token expected. 'BEGIN' reserved words are stacked because they may signal either a compound declaration or compound-statement. After all the declations have been processed SPTR is adjusted, if necessary, to point to the token which begins the first directive or statement.

(b) Sequence DECL calls either ITEM-DECL, TABLE-DECL or EXTERNAL-DECL to process a declaration.

4. Data Processor

A J73 program specifies data elements in data declarations and type declarations. It also specifies data operations by assignment statements; for example, the operations can be arithmetic or logical. When the control processor identifies a data operation, it activates the data processor.

The data processor DP directly executes the data constructs of the J73 language. It recognizes data and type declarations, creates data descriptors, and stores the data descriptors in the data associative memory DAM. The data descriptors in the DAM allow data references by symbolic names and permit rapid access of data values in the data memory. In addition, the DP executes assignment statements, evaluates

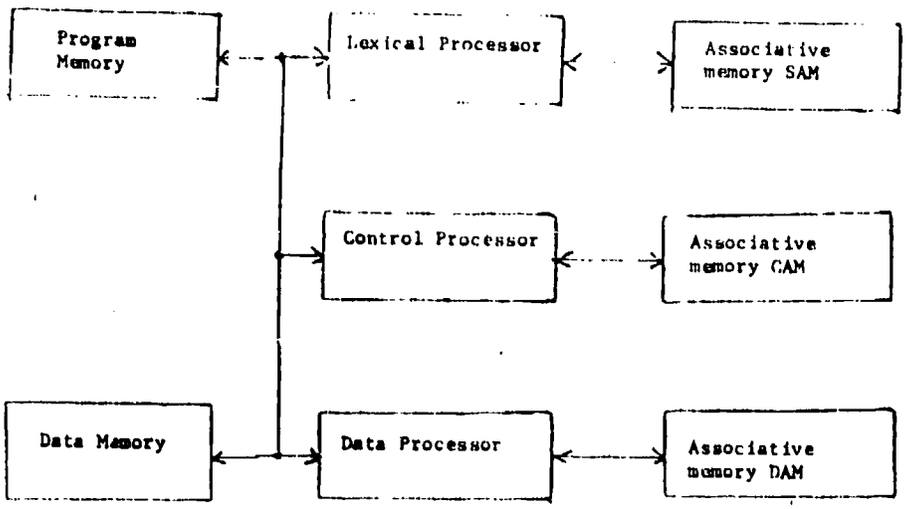


Fig. 2 Organization of a Direct Execution Computer

formulas, and handles parameters. The structure of the data processor DP consists of one associative memory, 1 register, and 5 stacks as shown in Fig. 5.

(a) The Data Memory stores the values of declared variables. One word of storage is allocated for each item or table element.

(b) Data Associative Memory DAM stores information about declared numeric variables. Name and Block-id fields identify each entry. Items and one-dimensional tables are the only possible structures. Possible types are signed and unsigned integer floating real numbers. The Size field identifies the number of DM words allocated for the variable. The trace-id field acts as a flag which identifies whether the variable is being traced.

(c) Register TRACE is a two-field register that serves as a flag to identify whether a variable is the object of a TRACE directive. The FLAG field is the switch, and the Name field saves the name of an assignment statement's variable for use in the output message which notes the variables new value.

(d) Stack SYNTAX contains the current syntax productions being executed.

(e) Stack VSTACK holds the value and type of formulas' operands and intermediate results. Operands must be items or table elements.

(f) Stack PSTACK holds the DM-locn of variables. The DM-locns of loop statement control variables are kept on the stack throughout execution of the loop statement, since the control variable is changed on every iteration.

(g) Stack OPSTACK saves lower precendence operators during evaluation of a formula.

(h) Stack APSTACK contains seven fields which save information about the actual parameters of a procedure call. Five of the fields holds the value, type, structure, size and parameter type (input or output) of a parameter. In addition, an output parameter's DM-locn is saved (so its value can be returned), and its name is saved if it is being traced.

(i) Stack FPSTACK has three fields to save information about an active procedure's formal parameter. The name and parameter type make up two fields. The third (Decld) is a flag which is set during the procedure's first execution if that parameter is declared in the module. All formal parameters must be declared. Also, the number, type and structure of actual and formal parameters must match.

The data processor DP processes data declarations and controls data flow. It is activated by CP, but it also communicates with LP. The data constructs that are processed by DP are:

(1) Directive
(2) items and table declarations
(3) assignment statement
(4) formulas
(5) boolean formula
(6) variable and subscript

(7) formal parameters
(8) actual parameters

4.1 Directives

The TRACE directive is a special statement which directs a message to be outputed whenever a variable in the statement's name list gets assigned a value. The syntax is:

17. <directive>:: = ! TRACE<name>,...;

4.2 Item and Table Declarations

Machine A accepts declarations in data, procedure, define, and block declarations. The CP executes define declarations. The DP executes data and block declarations. The syntax of declarations is shown below.

18. <item-declaration>:: = ITEM<name>(S!U!F);
19. <table-declaration>:: = TABLE<name> [<dimension>]·(S!U!F)·;
20. <dimension>:: = ·(<integer formula>)

The above syntax calls for the following hardware sequences.

01 ITEM...DECL

02 ITEM

01 TABLE...DECL

02 TABLE

02 DIMENSION

(2) Item Sequences

Item sequences consist of sequence ITEM... DECL and sequence ITEM which create an entry in the DAM from the name and attribute in the item-declaration and allocate a DM word.

(b) Table and Dimension Sequences

Table sequences consist of sequences TABLE ...DECL, TABLE, and DIMENSION. Sequences TABLE ...DECL and TABLE create an entry in the DAM. Sequence DIMENSION calculates the value of the dimension, which allows one dimension and only needs an upper bound (the lower bound is 0). This value is inserted into the size field, and a block of contingucus DM words equal to this value are allocated.

4.3 Assignment Statement

An assignment statement causes the value of a formula at the right of an equal sign to be assigned to the variable at the left of an equal sign. A variable is a name or a subscripted name. A subscript is an integer enclosed by a pair of brackets. The syntax is shown below.

21. <assignment-statement>:: = <variable>= <formula>

(a) Sequence ASSIGN-STMT

This sequence calls sequence VARIABLE to identify the object variable, and then calls sequence FORMULA to evaluate the formula. It then stores the formula's value into the DM location pointed to by the top entry of PSTACK.

(b) sequence TRACE-CHECK

Sequence TRACE-CHECK identifies whether the variable in an assignment statement or the output portion of an actual-parameter-list is being traced.

4.4 Boolean Formula

A boolean formula represents a value of TRUE or FALSE. It occurs in the IF-clause or the WHILE-clause. It can be either a formula followed by a relational operator further followed by a variable or a formula. The syntax is shown below.

24. <boolean-formula>::=<formula>[{!<> !<=!>=!<!>}<formula>]

The resulting value from a relational operation is either integer 1 for TRUE or zero for FALSE. The truth value of the boolean formula's result is determined by examining its low-order bit. A '1' is TRUE, '0' if FALSE. This implementation makes off integers evaluate to TRUE, even integer to FALSE.

4.5 Formula

A formula represents a value. It can be an integer formula or a floating formula, representing either an integer or a floating-point number, respectively. An integer formula is a positive or a negative integer term, which can be added to subtracted from a succeeding integer term. This intermediate result can then be added (or subtracted) to another integer term, and so on. (The arithmetic operators are left associative.) An integer term is an integer factor, which can be multiplied or divided by succeeding integer factors (as with terms). An integer factor can be an integer literal, a variable, or an integer formula enclosed by a pair of parentheses.

Floating formulas are similar to integer formulas, except a factor must be of floating type. In addition, a floating factor may be a call of function FLOAT, which converts an integer formula's value to floating form. The syntax for

(a) Actual Input Parameters

The actual input parameters can be 0 or more formulas. Each formula is evaluated; and its value, type, structure and parameter type are pushed onto AP-STACK.

(b) Actual Output Parameters

The actual output parameters can be 1 or more variables. Since each of the actual output

parameters that aren't tables set returned a new value, their DM-locns are also saved in the AP-STACK. Output parameters being traced also have their names placed in the Name field.

(c) Parameter Matching

Corresponding actual and formal parameters must agree in type, structure, size, and input/output type.

5. Lexical Processor

The J73 program is a string of characters. The lexical processor LP scans the characters in the source program, checks their legality, and assembles them into tokens. The tokens can be reserved words such as "ITEM" and "IF", operators such as "+" and ";", names, or numbers. The lexical processor together with the associative memory SAM also handles define declarations and define calls, and comments. It also handles the directive.

The structure of the lexical processor LP consists of an associative memory, and registers as shown in Fig. 5. They are described below.

(a) Program Memory PM contains the text of the JOVIAL program to be executed. It is arranged as one long string of characters. Each character is assigned an ordinal position so it can be identified by register SPIR.

(b) Scanner Associative Memory SAM stores information about define declarations. For each valid define declaration, an entry is created to store the name of the declaration, the location of the first character of the define body and the first after the last character of the define body.

(c) Table LEGALCHAR contains valid characters of the JOVIAL syntax and their respective classes.

(d) Table RESERWORD contains reserved words and their type. Special reserved words are 'DEFINE' (type 'D') and 'FLOAT' (type 'FLOAT'); the others are type 'R'.

(e) Register CHAR holds the last character fetched from Program Memory.

(f) Register CLASS holds the class of the character stored in register CHAR. The class, an interger, is found by searching the LEGAL-CHAR table.

(g) Stack RETURN saves the SPTR position immediately following a define call so that, after SPTR has advanced over the define body, it is reset to the proper position to continue program execution.

(h) Stack DEF-END saves the end-ptr positions of the bodies of active define calls. When SPTR reaches the position pointed to by the top entry of the stack, that define call is completed and a return is performed by popping DEF-END and popping return into SPTR. Recursive define calls are not allowed.

(i) There are two tables: LEGAL-CHAR and RESER-WORD. It needs to check each character of the source program to determine whether it is legal by looking up table LEGALCHAR. It needs to determine whether the new token is a reserved word by looking up table RESERWORD. The legal character table is shown in Table 2; there are 56 legal characters in 10 classes. The reserved word table is shown in Table 3; there are 19 reserved words.

The lexical processor LP scans the source string of characters, checks their legality, and assembles them into tokens. It is activated by either CP or DP. The lexical constructs are:

(1) token
(2) character
(3) name
(4) number
(5) operator
(6) define and comment

The hardware sequences of the LP which have sequence NEXT-TOKEN as the root sequence consists of:

```
01  NEXT-TOKEN
    02  NEXT-CHAR
    02  NAME
    02  DIRECTIVE-NAME
    02  DEFINE-DECL
    02  DEFINE-CALL
    02  REL-OP
    02  NUMBERICAL-LITERAL
        03  EXPONENT
        03  FRACTION
    02  COMMENT
```

5.1 TOKEN

Token is the lexical element of a source program. It can be a name, a number, an operator, or a separator as shown in the suntax below.

```
32. <next-token> ::= <name>
                   | <numeric-literal>
                   | <operator-separator>
                   | <reserved-word>

38. <operator-separator>  ::=
                ( | ) | : | ; | , | = | +
                | - | * | / | " | ' | . | <>
                | < | > | <= | >= | < | > | :
                | ; | , | " | ' | , | !blank

39. <reserved-word>  ::= START|PROGRAM|TERM
                         |BEGIN|END|ITEM|
                          TABLE|REF
                         |PROC|FOR|BY|WHILE|
                          IF|ELSE
                         | E|S|U|F
```

Sequence NEXT-TOKEN is designed to assemble the adjacent characters in the source program into a token. It extracts the next logical group of characters (the next token) from PM. The token

may be a reserved word, identifier, numeric-literal, operator, separator or directive. This sequence also handles define-declarations (macro definitions) and define-calls, because they affect the control flow of program text.

Initially, the starting position of the token is stored in register BACK-PTR. Then the token is formed. If the token is the reserved word 'DEFINE' a define-declaration is processed; if it is an identifier with an entry in the SAM a define-call is processed. When the next token to be passed to the other processors has been formed, the 'noise' following it is skipped over. Noise consists of blanks, illegal characters and comments. Upon return, the token will be in register TOKEN, its type will be in register TYPE, and register SPTR will be pointing to the beginning of the next token to be formed.

Sequence NEXT-TOKEN fetches the next char from the source program and then acts according to the class number of the character as follows.

class 1: An illegal char. Call ERROR.
class 2: A blank. Skip the blank.
class 3: A letter. The succeeding characters are assembled into a name. The name can be reserved word, an LP command, or an operand name.
class 4: A digit or period. The suceeding Characters are assembled into a number.
class 5: A decimal point. This case is handled the same as class 3.
class 6: Unary operator '+' or '-'. It is stored in register TOKEN.
class 7: An operator. It is stored in register TOKEN.
class 8: A '<' or '>'. A two-character operator ('<=', '>=', or '<>') is assembled.
class 9: A '!'. A directive name is assembled and identified.
class 10: A double-quote. A comment is flushed out.

5.2 Character

A character can be a letter, a digit, or a mark. There are 10 digits, 26 letters, and 17 marks, as shown below.

```
36. <character>  ::= <letter>
                   ! <digit>
                   ! <mark>

43. <digit>  ::= 0 ! 1 ! 2 ! 3 ! 4 ! 5 ! 6 !
                 ! 7 ! 8 ! 9

44. <letter>  ::= A!B!C!D!E!F!G!
                  !H!I!J!K!L!M!N!O
                  !P!Q!R!S!T!U!V!W
                  !X!Y!Z

45. <mark>    ::=+!-!*!/!:!<!=
                  !.!:!,!;!(!)!'
                  !.!!!blank
```

Sequence NEXT-CHAR fetches the next character from the source program in program memory. The next character is pointed to by register SPTR and becomes available in register CHAR. A test must now be made to determine if SPTR points to next to the end of a define body by comparing it to register DEF-END. If it does, SPTR is given the value of register RETURN (i.e. to return from the define call) before the next character is made available. The character is then tested for legality and register CLASS is set to the class number of the character.

5.4 Numeric-literal

A numerical-literal is a positive integer, and a floating literal is a numerical-literal with a decimal point. The lexical rules for numerical-literals and floating-literals are shown below.

```
34. <numeric-literal>  ::=<integer-literal>.
                         ! <floating-literal>

35. <integer-literal>::=<digit>...

36. <floating-literal>::=<digit>...
                           <exponent>
                         ! [<digit>].<digit>..
                           [<exponent>]

37. <exponent>       ::=E[+!-]<integer-
                          <literal>
```

Sequence NUMERICAL-LITERAL needs to detect the sequential combinations of digit, period, 'E', '+', '-', and others. There are 3 sequences as shown below.

```
01 NUMERICAL-LITERAL

    02 EXPONENT

    02 FRACTION
```

Sequence NUMERICAL-LITERAL constructs numerical-literals. There are two types: integer (type 'I') and floating (type 'FL'). Floating-literals have a decimal point and/or an exponent; integer-literals have neither. Sequence FRACTION extracts the digits following the decimal point of a floating-literal, while sequence EXPONENT extracts the exponent part of a floating-literal.

5.5 Relational Operators

The operators of machine a consist of single and double-character operators and the reserved words. Sequence REL-OP extracts the relational operators '<', '>', '<>', '<=', or '>='.

5.6 Comment

The comment is a string of 0 or more characters enclosed by a pair of quotes. The syntax is shown below.

```
35. <comment>::="[<character>]"
```

Sequence COMMENT flushes out the string of characters.

5.7 Define (Fig. 27)

The define-declaration is a macro definition; its body, like a comment, is a string of 0 or more characters. The syntax is shown below.

43. <define-declaration>::=DEFINE<name> "[<character>...]"

46. <define-call>::=<name>

Sequence DEFINE-DECL processes a define-declaration. The define-name cannot be the same as any name declared in the same module or any procedure name. A SAM entry is created to hold the name, module-id, location of the first character of the define-body, and location of the double-quote (') which signals the end of the define-doby for each valid declaration. The define-body is enclosed in double-quotes, so no comments are allowed between the define-name and define-body.

Sequence DEFINE-CALL processes a define-call. A define-call is not allowed when the name of a declaration or a procedure is the next token expected. On a valid call, the return location is saved by pushing it onto stack RETURN, register SPTR is assigned to point to the first character of the define-body and the end position of the define-body is pushed onto stack DEF-END.

The top of RETURN identifies the 'define-activation level' of the source program. This level needs to be known by the CP to determine if control statements may have CAM entries, so it is always stored in register D-LEV.

6. Concluding Remarks

The above JOVIAL Direct-Execution Machine A directly reflects the language constructs of the J73 language. The lexical processor directly recognizes the legal characters, reserved words, operands, operators. It assembles token, and executes lexical "commands" (such as the DEFINE constructs of the J73 language. The control processor directly executes the control statements and sequences the order of execution of the assignment statements; this control processor organization reflects the control constructs of the J73 language. The data processor directly references symbolic names and executes data operations; this data processor organization reflects the data constructs of the J73 language.

The above JOVIAL Machine A is a multiprocessor system; each processor performing a function reflecting language constructs. If the lexical processor were operated in a parallel but synchronized manner with the control processor and data processor, the repeated lexical processing in a program loop would not impede the execution speed. By using the information of the control structure of the source program in the associative memory CAM, there need be no repeated syntactical processing of the control statements in a program loop.

The idea of a direct-execution machine is simple, but its structure can be highly complex if the programming language such as JOVIAL is complex. Thus, there are two issues: the issue of the programming language and the issue of the computer architecture for the programming language. Criticisms on a particular direct-execution machine should address clearly the whether it is the language issue or it is to the architecture issue.

7. Acknowledgements

The author wishes to acknowledge the support of this work by Grant 79-0056 from the AFSOR/RADC. He also wishes to acknowledge help from several students, Marc Abrams, Eric W. Bonwit, Richard A. Britton, Edward Lor, Carmen Radelat, and Cliff Schaeffer, who helped in the preparation of the manuscript.

8. Reference

(1) Bloom, H.M., "Conceptual Design of a Direct High-Level Language Processor", Technical Report TR-239, Department of Computer Science, University of Maryland, April, 1973. (NITS PB-224098/AS.)

(2) Chu, Y., "Introducing the High-Level Language Computer Architecture", HIGH-LEVEL LANGUAGE COMPUTER ARCHITECTURE, Academic Press, Inc. 1975, pp. 1-4.

(3) Chu, Y., J.C. Yeh, and E.R. Cannon, "Direct-Execution on the Burroughs B1700 System", Technical Report TR-335, Computer Science Department, University of Maryland, Oct. 1974.

(4) Carlson, C.R., "A Survey of High-Level Language Computer Architecture", HIGH-LEVEL LANGUAGE COMPUTER ARCHITECTURE, Academic Press, Inc., 1975.

(5) Chu, Y., H.M. Bloom, and E.R. Cannon, "High-Level Language Hardware Control Architecture", Technical Report TR-412, Department of Computer Science, University of Maryland, Oct. 1975.

(6) Chu, Y., "Evalution of Computer Memory Structure", Processings of the National Computer Conference, June 1976, pp. 733-748.

(7) Chu, Y., "Architecture of a Hardware Data Interpreter", Proceedings of Computer

(8) Architecture Symposium, Silver Spring, Maryland, 1976, pp. 1-9.

(9) Chu, Y. and E. R. Cannon, "Design of a High-Level Computer Architecture", Technical Report TR-550, Department of Computer Science, University of Maryland, June 1977.

(10) Schwartz, Jules I., "The Development of JOVIAL", ACM SIGPLAN Notices, vol. 13, no. 8, Aug. 1978, pp. 203-214.

(11) "Military Standard JOVIAL (J73)", MIL-STD-1598A (USAF), Softech, Inc., March 15, 1979.

(12) Chu, Yoahan, "Direct Execution on a JOVIAL Machine", Technical Report TR-827, Department of Computer Science, University of Maryland, Nov. 1979.

(13) Chu, Yoahan, "Design of JOVIAL Direct Execution Machine", Technical Report TR-859, Deaprtment of Computer Science, University of Maryland, Feb. 1980.

```
00000     START
00100     PROGRAM TSTJOV;
00200        BEGIN
00300          REF PROC TRIG(ANG:SANG,CANG);
00400           BEGIN
00500            ITEM ANG U;
00600            ITEM SANG F;
00700            ITEM CANG F;
00800           END

00900                TABLE DEG[90] U;
01000                TABLE SSIN[90] F;
01100                TABLE CCOS[90] F;
01200                ITEM II S;
01300        ! TRACE DEG,SSIN,CCOS;
01400             FOR II:0 BY 1  WHILE II <= 90;
01500                 BEGIN
01600                  DEG[II]  =  II;
01700                 TRIG(DEG[II]:SSIN[II],CCOS[II]);
01800                 END
01900             END
02000     TERM
```

```
00000     START
00100     PROC TRIG(DEG:SIN,COS); BEGIN
                DEFINE PI "3.14159265";
00200
00300           ITEM JJ U;
00400           ITEM DEG U;
00500           ITEM RAD F;
00600           ITEM FACTR F;
00700           ITEM SIN F;
00800           ITEM COS F;
00900      "THIS PROCEDURE COMPUTES BOTH SIN AND COSINE OF AN ANGLE
01000                 USING A TAYLOR SERIES"
01100
01200
01300      SIN = 0.0;     COS = 1.E0;
01400      RAD = FLOAT(DEG)*PI/1.8E2;
01500      FACTR = 1.0;
01600      FOR JJ:1 BY 1 WHILE JJ <= 20; BEGIN
01700           FACTR = RAD * FACTR/FLOAT(JJ);
01800           IF JJ; BEGIN
01900              SIN = SIN + FACTR;
02000              FACTR = -FACTR;   END
02100           ELSE  COS = COS + FACTR; END  END
02200      TERM
```

Fig. 3 A Sample J73 Program

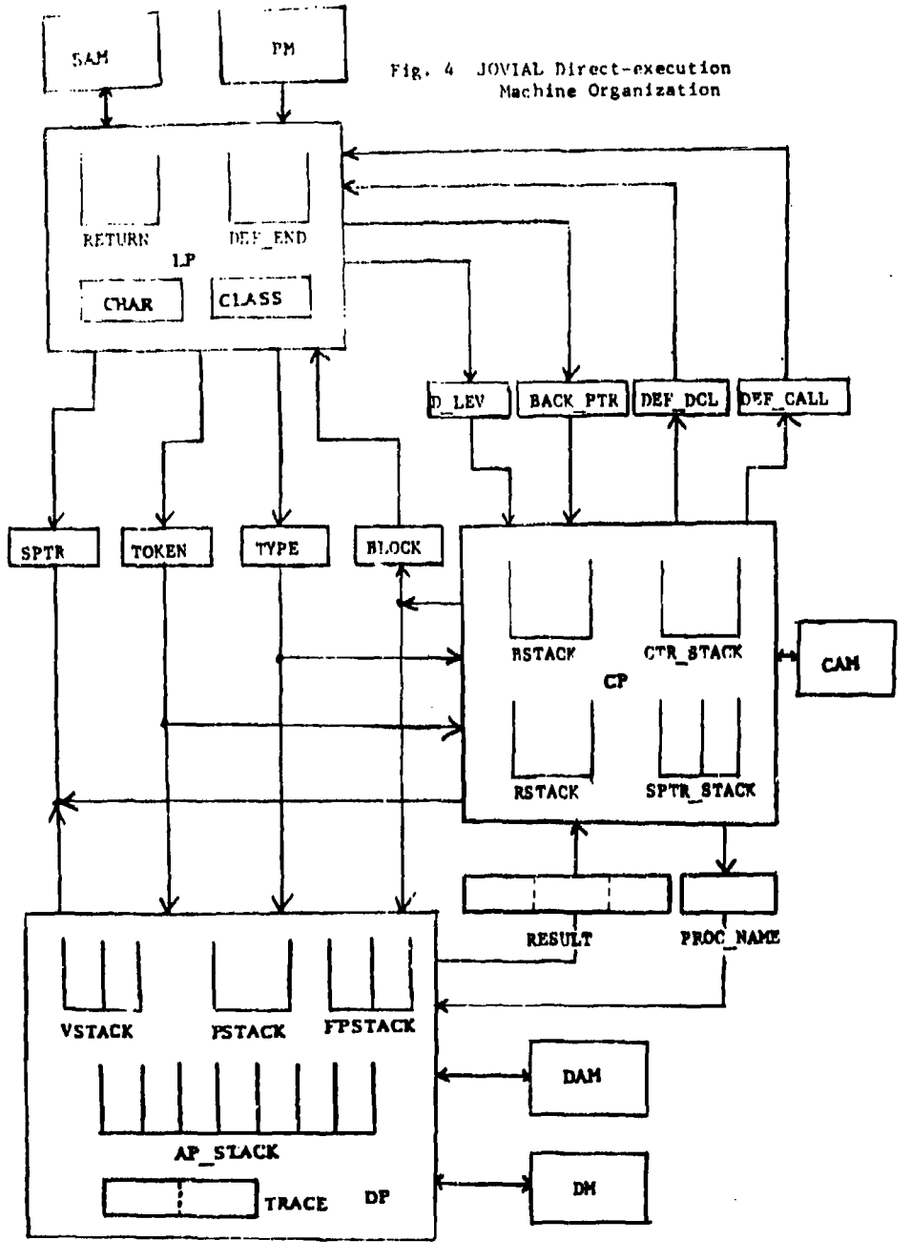


Fig. 4 JOVIAL Direct-execution Machine Organization

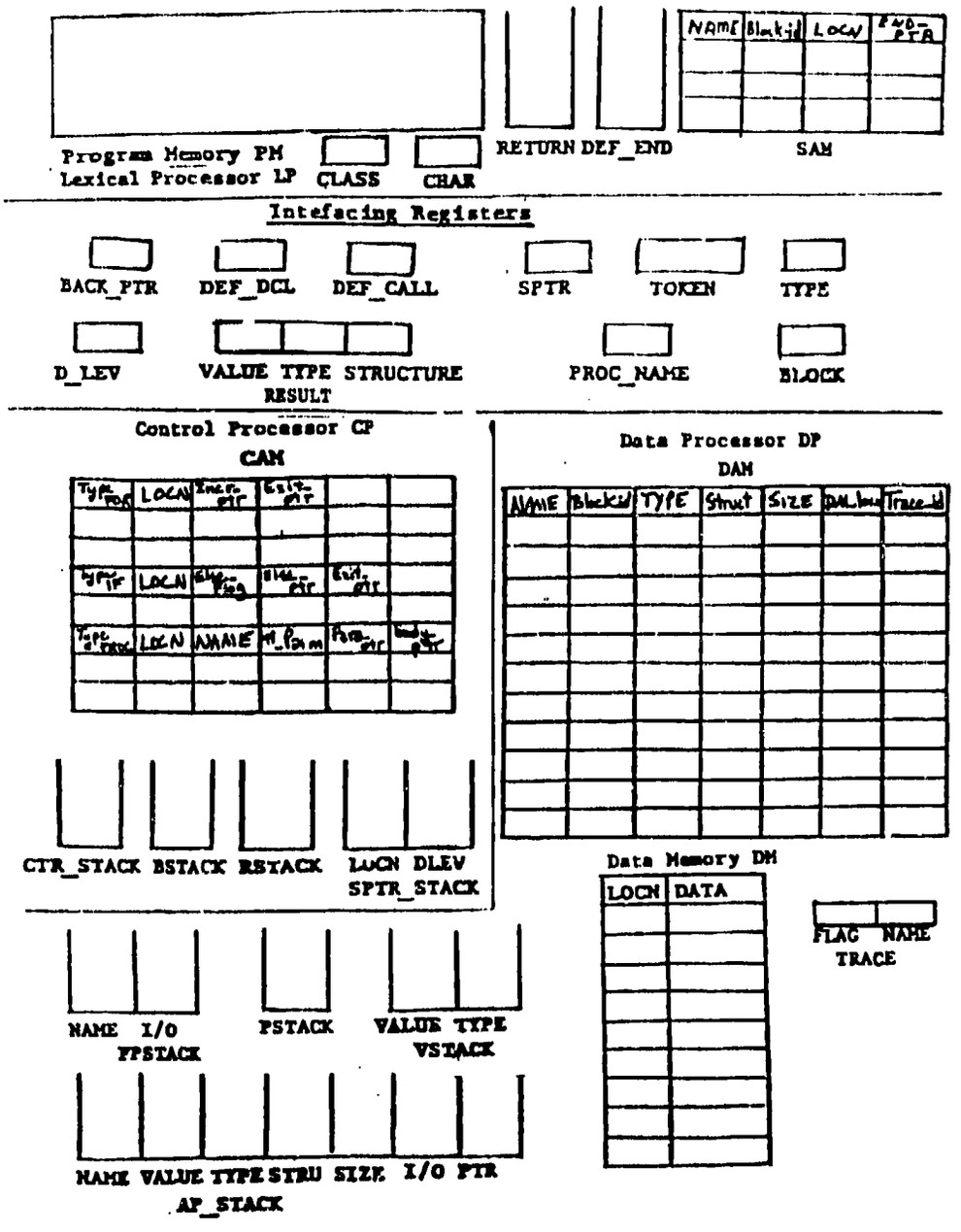


Fig. 5 Memory/register/stack structure of the JOVIAL Direct-execution Computer

QUEST FOR AN 'IDEAL' MACHINE LANGUAGE

Krishna M. Kavipurapu
(Southern Methodist University)

Harvey Cragon
(Texas Instruments)

Abstract

Researchers have realized that von Neumann machines do not adequately provide for the constructs that occur in common programming languages. Most of these shortcomings are attributable to a phenomenon known as semantic gap. Over the past decade there has been increased interest in building machines that have smaller semantic gap. It can be conjectured that there exists an 'ideal' directly executable language (DEL) which describes an architecture with a smaller semantic gap than conventional machines. The proof of this conjecture will enable us to evaluate candidate machine instructions and to select the most suitable machine language for a given computing environment. In order to prove this conjecture, certain characteristics of machines like the level of a machine with respect to a high-level language must be quantified. Halstead's Software Science metrics are used for this purpose.

Introduction

Before we start our introduction, we would like to define precisely the meaning of the term architecture as used in this paper. Computer architecture is the virtual machine as viewed by a machine language programmer. This is the view held by Flynn [75]. Thus, changing machine language (assembler language) changes the architecture. Using the same argument, all models of IBM/370 have the same architecture.

Researchers have realized that von Neumann machines do not adequately provide for the constructs that occur in common programming languages. Most of these shortcomings are attributable to a phenomenon known as semantic gap (Gagliardi [73]). The semantic gap is a measure of the difference between the concepts in high-level languages and the concepts in computer architecture. Most current systems have an undesirably large semantic gap in that the objects and operations reflected in their architecture are rarely closely related to the objects and operations provided in programming languages. As shown by Myers [78], this large semantic gap contributes to software unreliability, performance problems, excessive program size, compiler complexity and distortions of the programming languages, all of which contribute negatively to the economics of data processing.

The semantic gap can be reduced by constructing a high-level language machine for each language. Such high-level language machines have many advantages (Tannenbaum [76]). Over the past decade, there has been increased interest in building machines that have smaller semantic gap. These attempts are surveyed in Carlson [75] and Myers [78]. The proposed designs fall into 3 categories:

1. 'Truly' high-level language processors.
2. 'Pseudo' high-level language processors.
3. Intermediate language processors.

In 'truly' high-level language processors, (e.g. Bloom [73]) the processor accepts a program string written in a high-level language and performs operations as determined by the semantics of the program string. The important characteristic of this design is that the architecture operates on the program directly. A little thought will convince the reader that such a design is not the ideal alternative to von Neumann architectures from either the memory size standpoint or interpretation time standpoint (Hoevel [74]).

In 'pseudo' high-level language processors, (e.g. Burkle et.al. [78]) the source program is preprocessed; the software preprocessor performs a lexical transformation on the input changing the keywords and operators into internal code. All data objects in the program are replaced by references to memory locations. With the exception of superfluous blanks, preprocessing is an isomorphism.

The two high-level language processor designs described above are highly source language dependent and so a machine should be constructed for each high-level language. In the case of intermediate language processors, the source program is converted into a program in an intermediate language. The resulting surrogate program is executed by the architecture. It has been established (Wade and Schneider [73] and Lancaster [72],[76]) that a certain set of semantic primitives can adequately express the major portion of the semantics of programs written in any of the several common high-level languages. Therefore, it is conjectured (Wade and Schneider [73]) that by designing a computer organization which implements a set of semantic primitives that describe common high level constructs, one instruction per primitive, speed increases approaching that of a 'truly' high-level language processor can be achieved while retaining

the flexibility characteristic of software dominated conventional machines.

The authors believe that the intermediate language processor is the desirable choice. The authors also believe that there exists a direct relationship between the level of a target machine with respect to the source language (cf. SECTION 2) and the machine's dependence on the source language. That is to say, the higher the level of a machine with respect to a language, the more language dependent will the machine be.

Because of this relationship one can measure the closeness of a language to the machine. It can be conjectured that there exists an 'ideal' directly executable language (DEL) which describes an architecture with a smaller semantic gap than conventional machines. Hoevel [74] gave an analytical argument to show the existence of an 'ideal' directly executable language which performs better than conventional machines. In order to prove the above conjecture we must quantify certain characteristics of machines like the level of a machine with respect to a source language and semantic gap. This is the topic of present research. The metrics defined in this work are based on Halstead's (Halstead [77]) Software Science. This research is a step in the direction of quantifying architectures and is an attempt to bridge the gap between language designers and computer architects. The metrics defined can be used either to evaluate candidate intermediate languages and select the most suitable machine language for a given computing environment or to evaluate existing machines for a given environment.

Hoevel [74] has argued that neither machine language of conventional machines nor source language is an 'ideal' DEL either from interpretation standpoint or from storage point of view. He contends that an 'ideal' DEL for a contemporary computing system lies somewhere between its source language and the language accepted by its base machine. In this research, we attempt to prove that an 'ideal' DEL from semantic gap standpoint also lies somewhere between the source language and machine language. In the next section, software metrics that will be used to quantify architectures are defined. Results obtained so far are included.

Section 2

Halstead and his students (Halstead [73],[77] and Software Engineering [79]) found that application of the classical methods of natural sciences demonstrate that even such intangible objects as written abstracts and computer programs are governed by natural laws. Some of the metrics used by them that are pertinent to present work are now presented without explanation. Interested readers should refer to Halstead [77] for details.

1. The Volume V: A suitable metric for the size of any implementation of an algorithm, called the volume V, can be defined as

$$V = N \log_2 \eta \qquad (1)$$

where N is its length and η is the size of its vocabulary.

2. The Potential Volume V^*: The most succinct form in which an algorithm could ever be expressed would require the prior existence of a language in which the required operation was already defined or implemented, perhaps, as a subroutine or a procedure. The potential volume of an algorithm is the volume of the program which expresses the algorithm in its most succinct form:

$$V^* = (2 + \eta_2^*) \log_2 (2 + \eta_2^*) \qquad (2)$$

where η_2^* is the number of unique operands.

3. The Level of a Program L: Since there can be more than one possible implementation of an algorithm, it is necessary to define the level of a program. The level of a program L is defined as

$$L = V^*/V \qquad (3)$$

4. The Level of a Language Λ: When different algorithms are programmed in a given implementation language, it is observed (Halstead [77]) that as the potential volume V^* increases the program level L decreases proportionately. Consequently, the product L times V^* remains constant for any language. This product, the language level, is denoted by Λ:

$$\Lambda = L \cdot V^* \qquad (4)$$

The four quantities defined above form the basis of our research. In order to discuss the details a few more terms must be introduced.

5. Level of a machine with respect to a Language ∂_L^M: Certain machines are more closely related to the operations and data structures in a high-level language than other machines. A measurable quantity that describes this characteristic of a machine is in order. The level of a machine with respect to a language ∂_L^M is defined as

$$\partial_L^M = V_L/V_M \qquad (5)$$

V_L is the volume of an algorithm implementation in the language L and V_M is the volume in the machine language of the machine M.

Remarks: 1. The authors strongly believe that the quantity in equation (5) is a constant for a given machine M and a language L (and a compiler) and does not vary significantly with either algorithms or programming styles.

2. Compiler overhead is included while measuring volume V_M in equation (5). Thus, V_M is the volume of the program translated into machine language M by a compiler starting with the program in the high-level language L. This approach is used for practical reasons.

3. If compiler overhead is to be excluded, a different metric, the Potential Level of a Machine ∂_L^{*M} may be used:

$$\partial_L^{*M} = \Lambda_M / \Lambda_L \qquad (6)$$

where Λ_M is the level of the machine language of machine M and Λ_L is the level of the high-level language L. Potential level can be greater than 1 since it is possible to have a machine language whose level is higher than that of a high-level language. The performance of a compiler can be evaluated using the two levels defined above.

Some Results: 32 FORTRAN programs written by graduate and freshmen computer science students at SMU are used in our validation of equation (5). Operators and Operands in the programs used are counted according to the rules suggested by Bulut [74],[73]. The results are given in Table 1. When these values are plotted (Fig. 1), a straight line relation between the two volumes with a correlation coefficient of 0.978 is observed. From the plot, the level of Compass (assembler language of Cyber) with respect to FORTRAN (using FTN compiler with OPT = 0) is given by the slope of the curve:

$$\partial_{FTN}^{Compass} = 0.1716986$$

Similar computations are performed on COBOL and the results are tabulated in Table 2. The level of Compass with respect to COBOL is calculated to be (Fig. 2)

$$\partial_{COBOL}^{Compass} = 0.0517147$$

6. Dynamic Volume V_d: The volume of an algorithm defined in (1) is a static measure of the size of the algorithm and it can be used as an estimate of the memory required. However, the actual amount of code processed by the computer is different for different sets of data. Depending on the input, certain segments of the program may be executed more often than other segments. The Dynamic Volume of a program is the code of the program that is actually processed for a given set of data.

7. Average Information Rate I_M: Since it is possible to conceive of two machines with the same architecture where one machine executes programs faster than the other (e.g. the various models of IBM/370 series), a measure of the processing speed of machines must be defined. The average information rate I_M of a machine M is such a quantity and is given by

$$I_M = \frac{1}{\text{(Avg. time per run)}} \left[\frac{1}{T} \int_0^T V_d \cdot dT \right] \qquad (7)$$

where T is a sufficiently long time period over which the behavior of the program is observed, V_d is the dynamic volume of the program in the machine language and dT is the execution time of the program for the dynamic volume V_d.

Remarks: 1. To evaluate equation (7), a program (or a set of programs) must be executed with different sets of data. For each set of data, the dynamic volume V_d and the execution time dT must be noted. Then, the integration in equation (7) can be approximated by summation.

2. The product $I_M \cdot \partial_L^M$ is a measure of the speed at which programs written in a high-level language L are executed on machine M.

Some Results: A simple program is run on Cyber 72 a number of times with various values for input data. The values of execution time for various dynamic volumes are plotted in Fig. 3. As can be seen from the graph, the rate at which Cyber processes information is fairly constant and is given by the slope of the graph.

$$I_{Cyber\ 72} = 15.746 \times 10^6 \text{ bits/sec}$$

Applications

Although the results obtained so far are not enough to claim the validity of our metrics, they tend to support our intuition. However, since intuition is far from trustworthy, we are planning to collect data for three languages FORTRAN, Pascal, and COBOL and on three architectures Cyber, AMDAHL, and TI 9900. We believe that this set is a representative class of languages and machines most commonly used.

Once the consistency of these metrics has been validated, they can be used to select a machine language that is best suited for a computing environment. Denoting the set of programming languages under consideration by P, the machine language for which the quantity

$$\sum_{L \in P} (k_L \cdot \partial_L^M) \qquad (8)$$

is maximum describes an architecture with a minimum semantic gap for the set of programming languages P. The constant k_L in equation (8) is a weighting factor that reflects the frequency of usage of Language L in a particular environment. Typically, if 90% of the time COBOL is used in a given environment, k_{COBOL} will take a value of 0.9.

Equation (8) can also be used to evaluate existing architectures for a given environment. Use of the metrics defined in this paper provides useful information on the basic architecture of the machine and the implementation details such as the information processing rate are separated from the architecture. This information is not provided by benchmarks which reflect only the speed of execution of the benchmark programs on the machine. However, the authors believe that the counting techniques suggested by Bulut [74],[73] must be refined before existing architectures can be compared using our metrics.

Observations

While compiling FORTRAN programs, we tried various optimizations that are available on FTN compiler. After looking at the code generated, we decided to use only the code generated using FTN compiler with no optimization. The reason for this is the fact that optimization is not linear; only certain portions of the program are optimized. For example, no attempt is made to reduce the code required to implement subroutine calls and passing of parameters. Thus, if a program has a large number of subroutine calls, the amounts of code generated by both optimizing compiler and regular compiler are almost the same. This nonlinearity leads to an unfair comparison of FORTRAN programs.

We also observed that on Cyber 72, there are a few system Macros to execute most commonly occurring FORTRAN functions like format conversions for READ and WRITE statements. Similar observation can be made in connection with COBOL programs. So, the authors would like to stress the fact that the numbers obtained are for a virtual machine as viewed by a compiler writer. However, the use of such macros strengthens our belief that a new machine language which has a higher-level than conventional machine language is needed to improve the performance.

Conclusion

In this paper, the authors have attempted to introduce the subject of their research. The authors started out with an assumption that there exists an 'ideal' machine language which has most of the advantages of high-level language processors while retaining the flexibility of conventional von Neumann machines. In order to prove this conjecture, a few metrics are defined. Using these metrics, a most suitable machine language for a given computing environment can be designed.

Although, the actual values of our metrics may change if a different counting technique is used, the conclusions are still valid. The values obtained must be used only to compare two languages and no significance must be attached to the absolute values.

In our research, one basic assumption is that the language in which a program is written is the best language for that algorithm. However, we did not see any published results claiming the superiority of one language for a particular application. Our method can be extended to evaluate various programming languages for a given application. In order to do this, one has to write a number of programs (within a given area of application) in a set of programming languages and measure volumes of these algorithms in the different languages. The high-level language that has an overall minimum volume for the set of programs is the best implementation language for the area of application under consideration. Once again, we would like to caution the reader that the counting techniques may have to be refined before our method can be used for the suggested applications.

It is probably too early to outline the machine characteristics that cause semantic gap, but we observed that direct execution of a few high-level instructions would enhance the performance of computers appreciably. These instructions are very similar to the semantic primitives suggested by Lancaster [72].

Bibliography

Bloom, H. M. [73], "Structure of a Direct High-Level Language Processor," Proc. of Symp. on Architecture, ACM SIGPLAN Nov.

Bulut, N. [73], "Invariant Properties of Algorithms," Ph.D. Thesis, Purdue Univ., Dept. of Comp. Sci., Aug.

Bulut, N. [74], "Experimental Validation of a Structural Property of FORTRAN Algorithms," Comp. Sci. Tech. Rept., CSD TR 115, Purdue Univ., April.

Burkle, H. J., et.al. [78], "High-Level Language Oriented Hardware and the Post-von Neumann Era," ACM Computer Architecture News, April.

Carlson, C. R. [75], "Survey of High-Level Language Computer Architectures," in High-Level Language Computer Architecture, Edited by Chu, Y., Academic Press.

Flynn, M. J. [75] "Interpretation, Microprogramming, and the Control of a Computer," in Introduction to Computer Architecture Edited by Stone, H. S., SRA Inc.

Gagliardi, U. O. [73] "Report of Workshop 4 - Software Related Advances in Computer Hardware," Proc. of Symp. on High Cost of Software.

Halstead, M. H. [73] "Language Level - A Missing Concept in Information Theory," ACM SIGME, Performance Evaluation Review, March.

Halstead, M. H. [77] Elements of Software Science, American Elsevier.

Hoevel, L. W. [74] "Ideal Directly Executed Language: An Analytical Argument for Emulation," IEEE Trans. on Computers, Aug.

Lancaster, R. L. [72] "Semantic Primitives for Quick Implementation of a Family of Procedural Languages," Ph.D. Thesis, Purdue Univ.

Lancaster, R. L. and Schneider, V. B. [76], "Quick Compilar Construction Using Uniform Code Generators," Software - Practice and Experience.

Myers, G. J. [78] Advances in Computer Architecture, John Wiley & Sons.

Software Engineering [79] IEEE Trans. on Software Engineering, March.

Tannenbaum, A. S. [76] Structured Computer Organization, Prentice-Hall Inc.

Wade, B. W. and Schneider, V. B. [73] "A General Purpose High-Level Language Machine for Minicomputers," ACM SIGPLAN/SIGMICRO Interface Meeting.

$V_{FORTRAN}$	$V_{Compass}$	$V_{FORTRAN}$	$V_{Compass}$
2031.9704	6549.5468	449.4805	2821.7177
2347.1734	13846.4650	1946.9811	13816.5500
294.0315	1764.1995	3880.5540	23131.5030
421.1067	2408.9471	3110.9361	19617.8670
118.0280	770.3818	1596.1231	9580.6700
24.0000	304.2272	343.3762	1917.2798
361.2140	3006.6073	1153.7944	6561.7922
352.3330	2313.9810	1065.3293	8346.3600
375.0000	2957.3806	448.0130	1900.2761
208.1485	1843.3554	124.0000	805.3811
298.5551	2171.9507	73.0824	362.2120
385.0000	2788.0650	194.4868	1150.3680
55.3509	549.5718	246.3798	1623.9272
853.0376	5941.6165	12.0000	107.5489
128.0000	1149.2961	85.1101	886.8998
518.9212	3646.4257	1183.0721	6834.6131

Table 1. Validation of Equation (5)
FTN Compiler with OPT = 0 is used.

$V_{FORTRAN}$ is the volume in FORTRAN. $V_{Compass}$ is the volume in Compass.

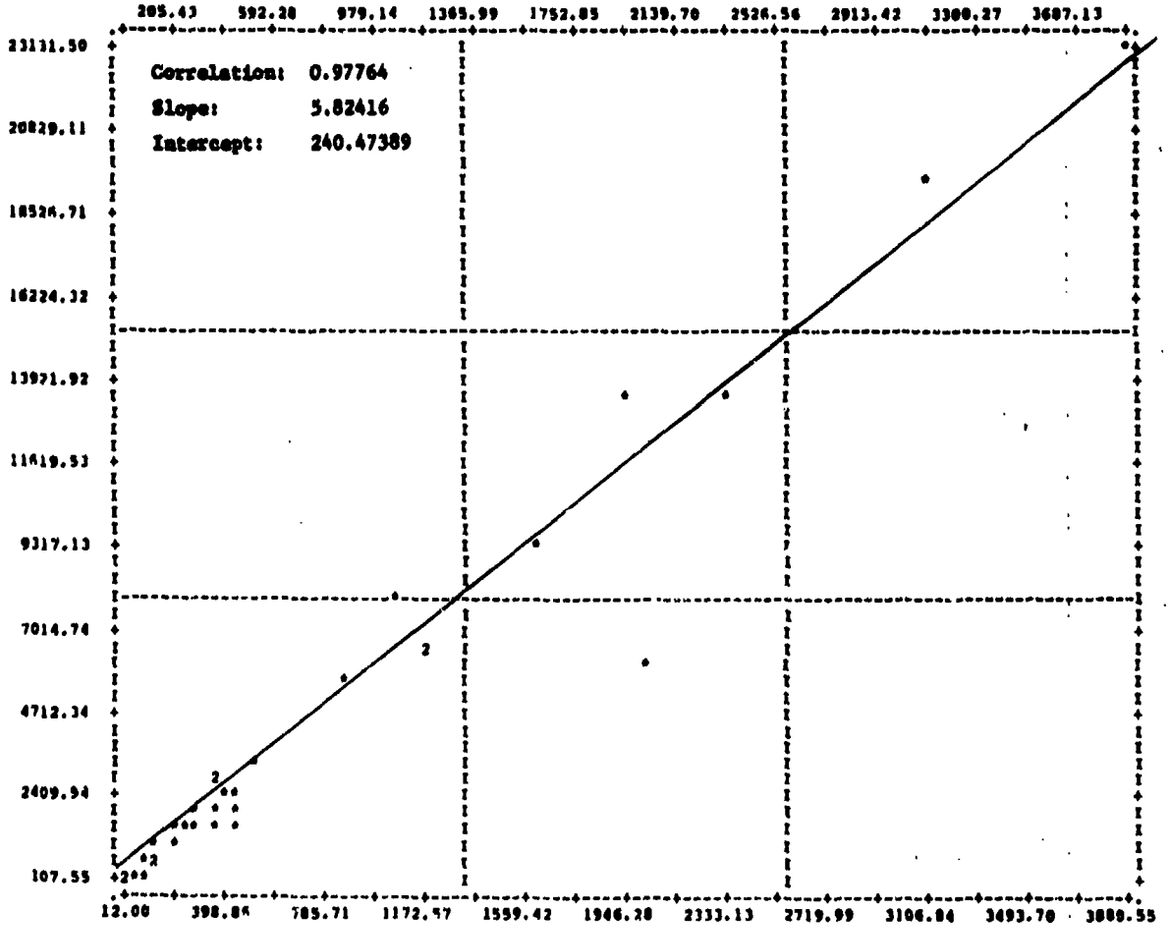


Figure 1. Validation of Equation (5)
$V_{FORTRAN}$ along X-axis. $V_{Compass}$ along Y-axis.

V_{COBOL}	$V_{Compass}$	V_{COBOL}	$V_{Compass}$
167.371790	3700.9530	227.548950	3977.9047
655.131790	11260.7840	230.321550	4645.9535
403.254150	10222.6950	483.308970	10455.0650
91.376518	2422.8076	21.000000	527.3324
46.506993	1250.2098	57.359400	1386.6956
122.984890	1951.2472	339.001500	7949.2895
245.969780	6198.6132	159.911340	3831.2925
286.620880	5333.9861	95.908275	2028.3122

Table 2. Validation of Equation (5)
COBOL compiler on Cyber 72 is used.

V_{COBOL} is the volume in COBOL. $V_{Compass}$ is the volume in Compass.

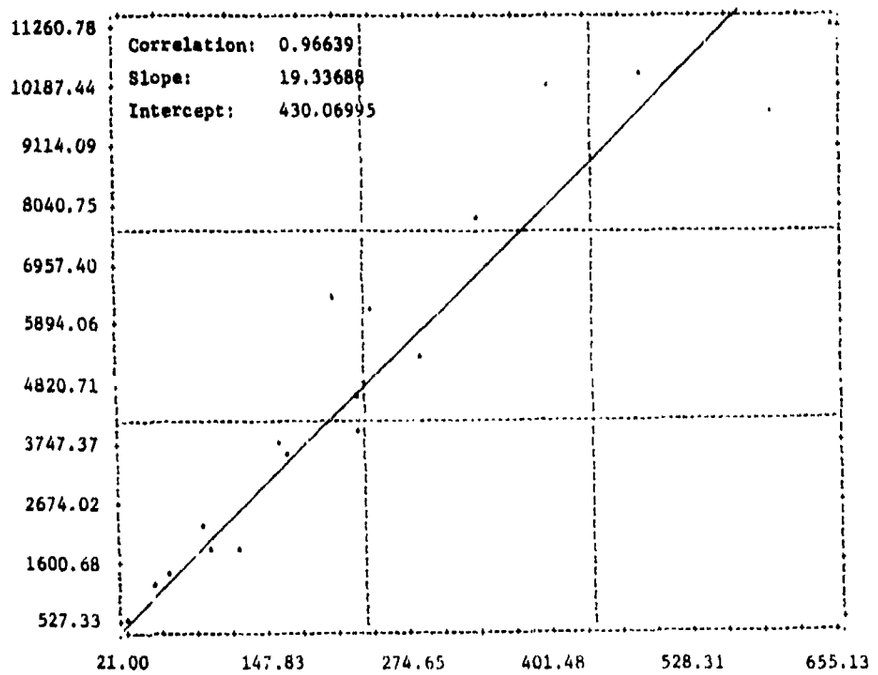


Figure 2. Validation of Equation (5)
V_{COBOL} along X-axis. $V_{Compass}$ along Y-axis.

V_d	dT	V_d	dT
5244252.049943	0.305	68172286.369940	4.285
10488254.909940	0.606	68172286.369940	4.245
15732257.769940	0.897	6737539.570993	0.353
20976260.629940	1.257	10106234.950990	0.506
26220263.489940	1.526	13474930.330990	0.676
31464266.349440	1.903	16843625.710990	0.844
36708269.209940	2.327	20212321.090990	1.012
41952272.069940	2.546	23581016.470990	1.178
47196274.929940	3.006	26949711.850990	1.341
52440277.789940	3.297	30318407.230990	1.522
57684280.649940	3.376	33687102.610990	1.680
57684280.649940	3.529	37055797.990990	2.021
62928283.509940	4.031	40424493.370990	2.177
62928283.509940	3.623	43793188.750990	2.317

Table 3. Validation of Equation (7)

V_d is the dynamic volume in Compass. dT is the execution time on Cyber 72.

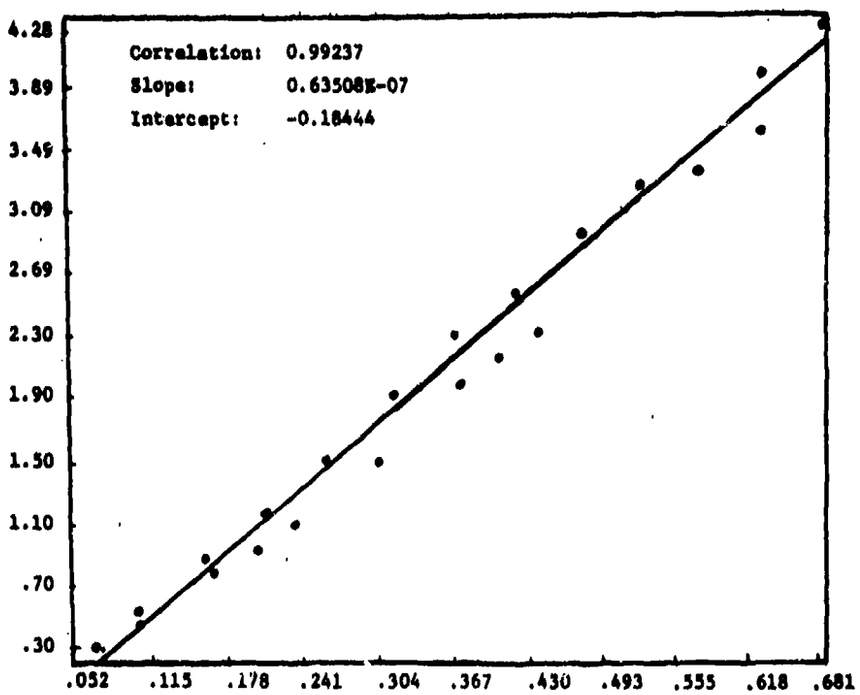


Figure 3. Validation of Equation (7)

$V_d \times 10^{-8}$ along X-axis. dT along Y-axis.

(units: V_d in bits, dT in seconds)

A DIRECTLY EXECUTABLE LANGUAGE SUITABLE FOR A BIT SLICE MICROPROCESSOR IMPLEMENTATION[1]

Neville R. Harris

Computer Systems Laboratory, Stanford University[2]

Abstract

Directly executed languages (DELs) as proposed by Flynn have variable sized fields for both operators and operands. For efficient implementation this architecture requires access to main memory at the bit level, and also requires powerful operations on varialbe sized bit fields in the host processor. For a hardware architecture based on bit slice processors and byte addressable memory it may be more advantageous to consider a byte oriented DEL. This simplifies the memory access hardware and makes the decoding of the DEL code a straight forward look up procedure. This paper reports on a project to build a Pascal oriented micro processor (POMP) and compares the POMP encoding of instructions with those of the DEL code. Initial results indicate that POMP code is less than fifty percent larger than DEL code and hence will be preferable when simplicity of interpretation is required.

Introduction

A Pascal oriented micro processor is being built at Trinity College Dublin using AMD bit slice processors. It will be used for research into the emulation of intermediate forms for block structured languages. Pascal will be the initial language considered and will also be used in all examples in this paper. During the design an architecture to efficiently support Flynn's DELs was considered. It would have required bit addressable memory, and operators for variable sized bit fields. Instead an architecture based on byte sized instructions was chosen to give easier interpretation and a simpler main memory interface. It was also felt that a compiler producing byte sized instructions would be easier to construct than one producing DEL code. The only disadvantage is the loss of compactness of code. This paper reports on initial investigations into the comparison of the two encodings and considers the tradoff between compactness of code and ease of interpretation.

[1] The work described herein was supported in part by the Army Research Office - Durham under contract no. DAAG29-78-0205.

[2] On leave from the Department of Computer Science, Trinity College Dublin.

Directly Executable Languages

The aim of DEL code [1,2,3] is to provide an ideal architecture for any high level language. The encoding is not optimum in the Huffman coding sense but instead a compromise between compact coding and ease of interpretation. "An ideal representation must be concise in its coding of identifiers yet not so concise that it exacerbates interpretation" [2 page 22]. In this architecture the scope of an identifier in a procedure is very important. The address of an identifier is given as the address (offset) within the contour. Hence, the number of bits required to hold an address is given by $\log_2$ (V) where V is the number of unique identifiers within the scope. Operators can also be encoded in this manner but the number of operators is small and hence a fixed encoding may be used instead. The DEL code instructions mirror the operations in the high level language giving three address type instructions. When the stack is required in expression evaluation then all loads and stores come as additions to the main operation being performed. In a sense they come for free. The loads and stores are not explicitly given in the DEL code instead they are implicitly applied as part of other operations. Thirty two formats specify all the different forms of the three address instructions. The DEL encoding for a number of expressions is given in figure 1. The encoding contains the format, operands and operations fields. In the format field A, B and C represent the three operands of an instruction when they are not in the stack. S represents the resulting operand pushed on top of the stack, and T represents the operand on top of the stack; which may be popped if required, and U is the next to top operand on the stack.

In examples 1 and 2 the operation is performed without the use of the stack. In examples 3 and 4 the stack is used and its use is indicated by the format of the instruction. In all the examples there is one DEL code instruction for each operator in the high level language expression. Note also that the load and store stack are implicitly implied by the format and combined with the instruction operation. One memory reference is saved in example 4 where the identifier K appears more than once. Conditional statements and the addressing of arrays can also be accomplished in a similar manner, as shown in figure 2.

The if statement in example 1 produces a DEL code instruction to test the condition and skip if the condition is not true, and another instruction to evaluate the expression K:= K + 1, which is executed, if the condition is true. In example 2 the array address calculation is considered as a single operation along with the assignment, and in example 3 it is considered as a single operation along with loading or storing from the stack.

This encoding produces compact code, anywhere from three to eight times more compact than that produced by compilers for traditional machines.

Pascal Oriented Microprocessor (POMP)

The size of procedures written in a structured manner, using a high level language tend to be small [4]. The most frequently occurring statement is assignment, followed by procedure call, if and return. Assignment statements tend to be very simple with the majority having only one or two terms on the right hand side. The majority of procedures have a small number of formal parameters and a small number of local scalar variables. Hence, the addresses of local variables and the most frequently occurring global variables may be compactly encoded. The coding of procedure calls must also be carefully considered.

A significant compaction of code can be gained from the fact that during the execution of any Pascal statement the state of the processor is always known e.g. integer or real. Between statements the state of the processor returns to the null state. For example the statements

```
var J,K : integer ; A,B : real;
    J := K + TRUNC(A + B)
```

produce the following instructions for a stack machine. The state of the processor is also given.

Instructions	Processor State
	=> Null
LOAD K	=> Integer
LOAD A	=> Real
LOAD B	=> Real
ADD	=> Real
TRC	=> Integer
ADD	=> Integer
STORE J	=> Null

The processor state is null between each statement and is set by load instructions and, in this example, by the truncate instruction also. Advantage can be taken of this fact [5] to provide a two dimensional instruction set thereby greatly increasing the range of opcodes available. Eight states of the processor are used.

0 - Null

1 - Boolean

2 - ASCII (Character)

3 - Address (pointer)

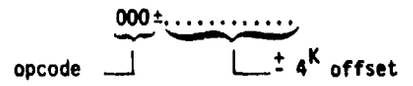
4 - Bit address (for packed structures)

5 - Integer

6 - Real

7 - Set

The state is contained in a three bit field in the processor's PSW. Some instructions are state independent, e.g. LOADS, and hence the opcode range is divided into a state dependent and a state independent range. Assuming that these ranges are equal in size then there are 1152 potential opcodes. For this architecture the low end of the opcode range is for state dependent instructions. The opcode range X'00' to X'2F' has been reserved for zero address instructions. The opcode X'10' represents integer addition if the processor state is integer and real addition if the state is real. The null and boolean states are used for unconditional jumps and false jumps respectively. A few lines from this area of the opcode table are shown in figure 3. Each opcode represents five different operations depending on the processor state. Branch instructions are implemented in both a short and long form. The short form is given in this area of the opcode table and consists of two bytes in the following form.



This requires thirty two opcodes X'00' to X'1F'. The long form jump consists of an opcode followed by a two byte offset.

Load instructions, which are state independent, are used to load the stack and also set the processor's state. Eight of these are provided for each of the states: boolean, ASCII, address, integer, real and set. Three bits within the byte give the local variable number and the format is

The long form of these instructions is used if the procedure has more than eight local variables - this will occur six percent of the time [4].

Separate one byte opcodes are used to perform operations between the top of the stack and eight local variables. If a local variable is added to the top of the stack this requires a one byte instruction rather than two instructions in the conventional stack machine. These instructions are two dimensional in that the meaning of the operation also depends on the processor state. The operations

involved are add, subtract, multiply, divide, compare for equality, compare for inequality and store. The null state of this part of the opcode table cannot be used with these operations and hence is used to zero local integer, increment local integer, and decrement local integer. These instructions also have eight different opcodes for eight local variables and again they replace either three or two instructions in the conventional stack machine.

Test Results

The code generator (the assembler) of the P code compiler was modified in order to obtain a feel for the compactness of the POMP code. The modified compiler produces either P code [6] or a combination of P code and POMP code depending on the setting of a number of control flags. These flags are used to test out the relevent importance of compacting different P code instructions rather than only obtaining the total effect. The P compiler produces code for a stack machine where all operations are performed on the top of the stack. Hence no advantage could be taken of the POMP instructions which operate between local variables and the top of the stack. For example the expressions A := B * C and A := A + 1 produce the following P code and POMP code:

```
          A := B * C

P code - LOAD B     POMP code - LOAD B
         LOAD C                 MUL  C
         MUL                    STA  A
         STA  A

       A := A + 1

P code - LOAD A     POMP code - INC A
         INC  1
         STA  A
```

The four areas most easily implemented and which were considered to result in the greatest compaction are:

1) Short branches - offset relative to PC

2) Loading and storing local variables

3) Loading small integers (0,1 and 2), loading boolean true or false, increment and decrement top of stack by 1

4) Zero address operators i.e. acting on the top two elements of the stack.

The P code compiler produces one P code instruction per 32 bit computer word. No compaction of the code was considered.

The results were then compared with the DEL code produced by a DEL compiler being implemented at the Stanford Emulation Laboratory. The object code size produced by compiling a quicksort program on the three different compilers were:

	DEL	POMP code	P code
Size	292	430	1004
Factor	1.0	1.47	3.44

The reduction in P code size due to the POMP instructions, broken down by compaction type were:

Compaction in bytes	%	Number of instructions	Compaction type
34	(6)	17	Short branches
267	(47)	89	Load and Store local variables
129	(22)	43	Load integers 0, 1, 2, load boolean true or false, increment and decrement by 1
144	(25)	48	Zero address instructions - operating on top 2 elements of the stack.
574		197	

The total number of instructions is 251 for both the P code and POMP code. In the POMP code they are broken down into 180 one byte instructions, 17 two byte instructions and 54 P code instructions. Almost half of the compaction is achieved by compacting the load and store locals. In contrast the short branches had almost no effect (6%).

From the preliminary results it looks improbable that the POMP code produced from the P code compiler can achieve the compactness of the DEL code. Each POMP instruction would on average only occupy 1.16 bytes as the DEL code for this program consists of only 68 instructions compared to 251 for the P compiler: a factor of 3.7. Even allowing for the fact that DEL operators often have implicit loads and stores associated with them there is still a remarkable difference in the number of operations. Hence the P code compiler has been discarded and present work is using a Pascal compiler which generates an abstract syntax tree during parsing. Using this compiler the full POMP code can be generated including the instructions which operate between local variables and the top of the stack. Statistics will also be generated on fourteen substantial Pascal programs giving the frequency of operators and memory references, and the resulting DEL and POMP codes will be compared.

An advantage put forward for minimizing the number of instructions of the object code is that it speeds up the execution. A large number of instructions increases the fetch and decoding time but with instruction prefetch and with simple ROM look up decoding it is expected that the difference in execution speed due to this effect will be small.

References

[1] Hoevel, L.W., and Flynn, M.J., "The Structure of Directly Executed Languages: A New Theory of Interpretive System Design," CSL Tech. Rpt. 130, Stanford University, March 1977.

[2] Flynn, M.J. and Hoevel, L.W., "A Theory of Interpretive Architectures: Ideal Language Machines," CSL Tech. Rpt. 170, Stanford University February 1979.

[3] Hoevel, L.W. and Flynn, M. J., "A Theory of Interpretive Architectures: Some Notes on DEL Design and a Fortran Case Study," CSL Tech. Rpt. 171, Stanford University, February 1979.

[4] Tanenbaum, A.S., "Implications of Structured Programming for Machine Architecture, CACM Vol. 21, No. 3, March 1978.

[5] Jones, J., "Towards a High Level Language Microprocessor Oriented Instruction Set," (to be published - Euromicro). 1980.

[6] Alpert, D., "A Pascal P Code Interpreter for the Stanford Emmy," CSL Tech. Note 164, Stanford Universit September 1979.

	Expression	Format	Operands	Operation	Stack Value
1)	K := J + 2	ABC	J 2 K	+	-
2)	K := K + 2	AAB	K 2	+	-
3)	D := K + J + L + 7	SAB	K J	+	K + J
		TTA	L	+	(K + J) + L
		ATB	7 D	+	-
4)	D := K + K + K + 7	SAA	K	+	K + K
		TTA	K	+	(K + K) + K
		ATB	7 D	+	-

Figure 1

	Expression	Format	Operands	Operation	Stack
1)	if K = J then	SAB	K J Skip offset	< > GoTo	-
	K := K + 1	AAB	K 1	+	-
2)	K := H[J]	ARRAYA	J H K	A := B[X]	-
3)	H[K] := [J] + N[J]	ARRAYA	J M	S := A[X]	M[J]
		ARRAYA	J N	S := A[X]	N[J],M[J]
		TUT		+	M[J] + N[J]
		ARRAYA	K H	A[X] := T	

Figure 2

Opcode	Null	Bool	ASCII	Addr	Bit.Add	Int	Real	Set
X'10'	UJP	FJP	-	-	-	ADI	ADR	UNI
X'11'	UJP	FJP	-	-	-	SBI	SBR	DIF
X'12'	UJP	FJP	-	-	-	MPI	MPR	INN

Figure 3

PARTIAL EVALUATION OF A HIGH-LEVEL ARCHITECTURE

Göran Bäge, L M Ericsson, S-126 25 Stockholm, Sweden[§]

Lars-Erik Thorelli, Department of Telecommunication and Computer Systems,
Royal Institute of Technology,
S-100 44 Stockholm, Sweden

Abstract

The architecture of the high-level language machine LAX2, designed for efficiency in string manipulation and interactive applications, is evaluated with respect to program volume and number of interpreted instruction bits. The evaluation takes the form of a comparison with the PDP-11 architecture using av test data a set of complete, realistic programs from a well-known source. The result shows the superiority of the high-level architecture.

Introduction

LAX2[1,2] is a high-level architecture designed to be efficient for string manipulation and interactive applications. It has type-marked values, dynamic storage allocation, and powerful instructions for string manipulation. The language of the machine is specified in two levels, a source or text level TLAX and an executable level ELAX. There are no GOTO's in TLAX; all jumps are generated from high-level control structures by the simple TLAX - ELAX compiler which is a fixed part of the machine. Memory is splitted into a number of data and program blocks; relative and indirect addressing is used with out-of-bounds checking to achieve compact code and high reliability.

The main design goals for LAX2 are low cost for software production and good memory and execution time economy for the intended class of applications. The design has been heavily influenced by the concepts of structured programming. The architecture has been implemented as a partially microcoded interpreter on a Varian V73 minicomputer.

The present paper reports on an evaluation of the LAX2 architecture. The evaluation is only concerned with memory and execution time economy, leaving out completely aspects such as ease of programming and debugging, software security, and ease of compilation. Furthermore, the number of interpreted instruction bits, rather than physical execution time, is used as the dynamic measure. The evaluation consists of a comparison of LAX2 with PDP-11, using a set of programs taken from the well-known book Software Tools by Kernighan and Plauger[3].

It turns out that LAX2 uses significantly fewer bits for instructions, both statically and dynamically. Thus, the present study gives yet another example of the superiority of high-level architecture, designed from language and application considerations, over conventional architecture. After a short description of the high-level architecture the evaluation method and results are presented. The concluding sections compare the present work with earlier evaluation studies and discuss the significance of the results.

Short description of the high-level architecture

LAX2 is a tagged architecture[4]. Its design presupposes a basic word format of 16 bits. Currently the machine recognizes types of values according to Figure 1.

```
Simple types:  nil, boolean, character, index
               (integer in the range 0-16383)
Composite types:
   string      (of characters)
   node        (heterogeneous array)
   decimal     (decimally represented integer)
   prog        (executable procedure)
   coprog      (coroutine activation)
   channel     (for input or output)
   (real, realarray planned, not yet implemented)
```

Figure 1. LAX2 data types

A value of simple type is represented by one 16 bit word with its leftmost bit cleared. A composite value is represented by a 16 bit word, the head, whose leftmost bit is set, pointing to a memory block, the body, containing a type-and-length descriptor and the value proper.

The memory area of a LAX2 process is divided into a stack in which procedure activation records are allocated, and a heap, where compactifying garbage collection is performed when necessary (Figure 2).

[§] The work was done while the author was with the Group for Datalogical Research, Stockholm

Figure 2. Memory area of LAX2 process

An executable procedure, i. e. a prog value, can only be created by means of the LAX2 instruction 'compile', taking a string, the TLAX version of the procedure, as main argument. The body of a prog value is shown (with some simplification) in Figure 3. Each rectangle represents a 16 bit word.

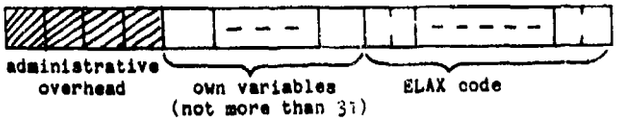


Figure 3. A prog value

The ELAX code can only access the own variables and stack variables (locals and parameters) of the current activation record. Figure 4 shows the structure of an activation record on the stack. The stack variables are also represented by one word each, and their number may not exceed 32. In this way addresses to commonly referenced quatities are kept very short. More remote information is reached through indirect addressing. A complete user program consists of a network of prog and data values linked by the own variables of the prog's.

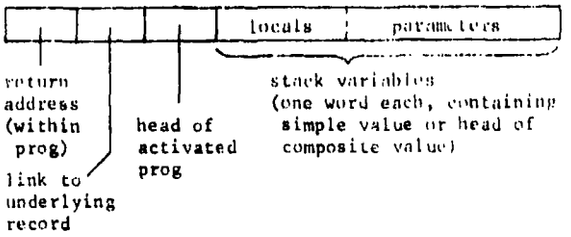


Figure 4. Activation record

Dynamic type checking and the static checking performed by the 'compile' instruction catch a great number of possible programming errors. Another feature promoting the efficient production of reliable software is absence of jump instructions in the TLAX representation. All jumps are generated during compilation from high-level control structures. In many other respects TLAX offers a rather primitive notation which, together with the high level of ELAX, makes the compilation process simple.

ELAX code consists of a sequence of 8 bit bytes. The design is similar to that of EM-1 (Tanenbaum[4]) and is characterized by compactness and the possibility of fast instruction decoding. Figure 5 shows some simple statements in Algol-like notation and their ELAX counterparts.

Statement	ELAX code	No of bytes
A:=B+3	push B, push 3, add, locate A, store	5
A:=A-B	push B, locate A, minus	3
A:=A+1	locate A, incr	2
A:=0	locate A, clear	2

Figure 5. Simple ELAX examples

LAX2 has a powerful set of string manipulation instructions. A small example is given in Figure 6. The guiding principle has been that although it should be simple to dynamically create and throw away strings, this feature should not be forced upon the programmer, and that lexical and other kinds of string analysis could be done with high machine efficiency. The reader is referred to (1,2) for further information on this and other aspects of the LAX2 architecture. Appendix A summarizes the ELAX instruction list.

Problem: The string S contains an identifier, an operator symbol and an unsigned integer, possibly separated by blanks. Assign the identifier (a string) to A, the operator (a character) to OP, and the integer (an index) to B.

ELAX solution: locate V, clear,
push S, locate V, getident, locate A, store,
push S, locate V, getchar, locate OP, store,
push S, locate V, getindex, locate B, store.
Total number of bytes: 17

Figure 6. String analysis example

Method of evaluation

The book Software Tools[3] is highly suitable as a source of benchmark programs for LAX2, since the programs are complete, have been used in practice, and are typical of the application area of the machine. The programming language used in (3) is Ratfor, a structured dialect of Fortran. The following programs were selected for use in the investigation.

a. ENTAB ((3) pp 37,21,20).
Copies a text file, substituting each sequence of spaces preceding a tab stop by a tab character. Tab stops are located at each 8'th position in the line.
Program size: 46 lines of source code (not counting comment and blank lines).

b. COMPRESS ((3) p 44).
Produces a compressed version of a file using run length compression, i. e., a sequence of identical characters is encoded by length and character value.
Program size: 36 lines.

c. PUTDEC ((3) pp 61,62, plus a main routine).
Converts integers to ASCII format and places them in specified fields.
Program size: 38 lines (excl. main routine).

d. QUICKSORT ((3) pp 115,110,111, plus a main routine).
Sorts a sequence of text lines into lexicographical order by means of the well-known "quicksort" algorithm.
Program size: 66 lines (excl. main routine).

e. FIND ((3) pp 136-138).
Searches a file, outputting each line containing a certain pattern given as input. The pattern is essentially a regular expression.
Program size: 279 lines.

The set of programs is rather small but is hoped to be representative of the text processing application area.

Next, these programs were translated for the two architectures LAX2 and PDP-11.

The translation for LAX2 was obtained as follows. The programs were rewritten into the language HLAX, a high-level (above TLAX) notation for LAX2. The HLAX programs were compiled using a cross-compiler on a DEC-10 computer. During the rewriting process care was taken to stay close to the original programs. As a consequence the programs run on LAX2 have, except for minor details, the same data and program structures and use the same algorithms as the original programs. This means that the features of LAX2 have not been used to full advantage. However, an additional version (FIND.OPT) of FIND, optimized for LAX2, was written. The optimization relies mainly on the observation that a majority of search patterns consist of or start by a literal string. Therefore it should pay to modify the internal representation of patterns and use the substring searching 'part' instruction of LAX2. Also, a recursive version (QUICK.REC) was written in addition to the non-recursive version from (3).

To translate the programs to PDP-11 code the language C (6) was used. As before, the rewriting was done to faithfully preserve the given algorithms and structure. To obtain high quality machine code all features of the C language promoting this goal were used, including the possibility of declaring quantities to reside in registers. The programs were compiled using the optimizing compiler available under UNIX'. As a result of these measures we believe that the machine code is as efficient as that produced by a competent assembly language programmer, with the possible exception that the latter may in some cases feel inclined to use a less general subroutine calling sequence, to save time for the saving and restoring of registers. In addition to the five programs from (3) the recursive sort program QUICK.REC was produced also for PDP-11.

The results

The volumes of the programs, excluding input/output routines, were measured. The volume is defined as the size of the executable form of the program including statically allocated data. The sorting programs operate on data in primary storage; this space is not included, as its size depends on the size of the input.

The result is displayed in Table 1.

Program	Result PDP-11	Result LAX2	LAX2/PDP11 in %
ENTAB	204	189	93
COMPRESS	251	210	84
PUTDEC [1]	97	60	62
QUICKSORT [1]	233	132	57
QUICK.REC [1]	175	77	44
FIND	1282	776	61
FIND.OPT	(1282)	841	66
Total	2242	1444 [2]	64

Notes: 1: excluding main program
2: excluding FIND.OPT

Table 1. Program volumes (unit: 16 bit words)

The high percentage figures for the first two programs are explained by the fact that they use data structures whose sizes dominate over the sizes of the programs proper.

In addition to these static results, dynamic measurements were derived. The programs were executed on the two machines and the number of interpreted instruction bits was recorded. These counts exclude all input/output handling.

The following text files were used for input during the dynamic measurements (Table 2).

File	Content	No of ASCII symbols	No of lines
TEXT0	extract from report	580	13
TEXT1	extract from report	4752	98
TEXT2	extract from report	3806	99
TEXT3	source code, C language	1038	51
TEXT4	mail address list	5697	100
TEXT5	mail address list	1139	20

Table 2. Input data

The ENTAB and COMPRESS programs were run using files TEXT1 - TEXT4 as input. PUTDEC uses no input file; instead, the main routine makes 36 calls on the conversion procedure.

The sorting programs were used to sort the lines of TEXT1, TEXT3, and TEXT4, and also an already sorted version TEXT4S of TEXT4, resulting in worst-case performance.

Finally, the FIND programs were run using a collection of 19 search patterns and the input files TEXT0, TEXT3, and TEXT5. Three groups of measurements were performed. Group 1 uses simple search patterns consisting of single literal strings. Group 3 uses complicated search patterns, and group 2 falls in between groups 1 and 3.

As in the selection of the test programs themselves, the aim in the selection of test data was to achieve realistic and typical conditions with a reasonable amount of effort.

Table 3 summarizes the result of the dynamic measurements. For FIND measurements were taken separately on the pattern building part (PATTERN) and the pattern matching part (MATCH).

The superiority of the high-level architecture is evident. Summing all measurements (omitting the non-recursive QUICKSORTs), we get the overall figure 28% for the ratio of LAX2 to PDP-11. However, the variation across the programs is high, and the result depends on the test data used.

The case of the optimized MATCH shows highly favourably values for LAX2, especially for group 1. The main explanation is that the search patterns of group 1 consist of single literal strings, allowing the search to be performed by the substring searching 'part' instruction. Likewise, the patterns of group 2 consist of literal strings appended by other constructs, so part of the search can be speeded up as in the case of group 1. The sorting

Program	Data	Result PDP-11	Result LAX2	LAX2/PDP11 in %
ENTAB	TEXT1-4	7105	4357	61
COMPRESS	TEXT1-4	9599	7308	76
PUTDEC	-	130	46.3	36
QUICKSORT	TEXT1,3,4	2942	421	14
QUICKSORT	TEXT4S	6000	558	9
QUICK.REC	TEXT1,3,4	2925	354	12
QUICK.REC	TEXT4S	5925	516	9
FIND: PATTERN	all patterns	667	187	28
FIND: MATCH	group 1	8323	2768	33
	group 2	16588	4778	29
	group 3	23286	6166	26
	group1-3	48197	13712	28
FIND.OPT: PATTERN	all patterns	(667)	121	18
FIND.OPT: MATCH	group 1	(8323)	68.8	1
	group 2	(16588)	1102	7
	group 3	(23286)	6630	28
	group1-3	(48197)	7801	16

Table 3. No of interpreted instruction bits (unit: 1000 bits)

programs also perform better than average on LAX2, mainly due to the use of string comparison instructions built into LAX2.

The least favourable case for LAX2 is the COMPRESS program. A closer look shows that this is the program with the lowest frequency of procedure calls. Procedure calling is more efficient in the high-level machine than in PDP-11, and the generality of the call-return sequence produced by the C compiler emphasizes the difference. Code optimization across procedure boundaries can be expected to improve the PDP-11 results in some cases. Such optimization is however a complex task.

Discussion

An objection to the results presented is that the influence of data storage and accessing has been neglected. Additional questions may be raised concerning the relevance of the number of

interpreted instruction bits as an architectural measure. These issues will now be discussed. In addition, the present work will be related to similar published investigations.

The volumes displayed in Table 1 do not include storage allocated dynamically during execution. How would this dynamic storage requirement affect the comparison? A look at the test programs shows that the effect is small. Only the sort programs can allocate more than in the order of 10 words. The sort programs use one more word per line of input in LAX2 than in PDP-11, due to the use of type-and-length descriptors. With the test data used this amounts to a 6% increase in data storage. The stack frames in LAX2 are smaller than those used by the PDP-11 code. The influence of this difference is small, however. The recursive sort programs grow a stack whose depth is only log2(n) frames, where n is the number of input lines.

The number of interpreted instruction bits has been shown to be small for LAX2 (Table 3). It might, however, be suspected that the number of memory references during access to data is higher for LAX2 than for the conventional machine, since each composite value is equipped with a one-word descriptor. Unfortunately no mechanism was available for monitoring this effect. Inspection shows that in the case of the test programs the descriptor references would add well below 5% to the execution time. The actual figure is of course quite implementation dependent, for instance, a cache memory would probably almost eliminate the overhead.

The number of interpreted instruction bits (NIB) is an architectural measure clearly related to execution speed. Small NIB values means that little time is spent in fetching instructions, however, the complexity of the decoding process must also be considered. In the case of LAX2 vs PDP-11 the latter factor seems to be of small importance.

Given a physical implementation of an architecture, one would expect the execution times of programs to be proportional to their NIB values. However, the accuracy of this correspondence depends on the homogeneity of the instruction set, i e the degree to which the instructions all "do the same amount of work". In particular, the effect of vector instructions has to be taken into account. Like many other high-level architectures LAX2 has instructions operating on variable length data, in particular strings. If such iterative or vector instructions are used frequently and on large data items, then clearly the number of interpreted instruction bits will give a too optimistic view of physical execution time.

To estimate this effect the use of vector instructions in the test programs was investigated. The programs ENTAB, COMPRESS, and PUTDEC make negligible use of vector instructions. The sorting programs compare text lines by means of the vector instruction 'string compare'. With the test data used the average number of iterations (character comparison steps) performed per such instruction is as low as 3. The relative frequency of the instruction is 5%. Let us assume, rather arbitrarily, that each iteration counts as three normal, i e non-vector, instructions. Assume further that all instructions are of the same length in bits - which is close to being true. Then we arrive at a prolongation factor of 1.4 due to the use of vector instructions. That is, to get a more realistic measure of expected execution time, add 40% to the results in Table 3 in the case of the sort programs.

The non-optimized FIND program makes less frequent use of vector instructions. The optimized FIND.OPT:MATCH, however, uses the substring searching instruction 'part'. For literal string patterns (group 1) we find the relative frequency of 'part' to be 2% and the average number of iterations to be close to 50. This gives a prolongation factor of close to 4.

These findings correlate well with the results of Table 3 but do not fully account for the high superiority of LAX2 in the cases discussed. The remaining cause seems to be that the PDP-11 versions are more heavily burdened with subroutine linkage than the LAX2 versions, where certain subroutines have been replaced by vector instructions.

As mentioned in the Introduction the LAX2 machine has been implemented as a partially microcoded interpreter on the Varian V73 minicomputer. The volume of the microcode is 180 64-bit words, and the remainder of the interpreter consists of approximately 7K 16-bit words of V73 machine code. Thus the microcoded part is small. Execution times on the two machines were measured for the test programs. The execution time ratio of LAX2-V73 to PDP-11/45 varies from 14 to 0.3 with 6-8 as typical values. These figures are quite satisfactory, considering the usual slowdown due to software interpretation. The hardware characteristics of the two minicomputers are roughly equal.

Finally a comparison of the present work with similar published investigations.

Wilner[8] has evaluated voulmes of Fortran and Cobol programs on Burroughs B1700 using language-oriented instruction sets, in comparison to IBM System S/360 (and Burroughs B3500). The results show improvements by a factor 2 to 3, larger than the factor of about 1.5 for LAX2 compared with PDP-11. This is however not surprising; S/360 code is less compact than PDP-11 code[9].

Wortman[10] compared the Student PL Machine of his own design with the S/360. A large number of small student programs were used as test cases. Several dynamic and static measures were evaluated. The results show a twentyfold superiority for his machine in number of instruction bits, both in the static and dynamic sense. However, it should be noted, first, that his S/360 programs were produced by the standard PL/I(F) compiler, and secondly, that all runtime checks built into his high level architecture are also included in the S/360 versions. These checks include the PL/I(F) conditions 'subscript range','overflow', and 'stringrange'. This is in contrast with our investigation, where such checks are indeed performed by the LAX2 machine but not by the PDP-11 program versions.

Nielsen[11] compared a proposed high-level language architecture for the SPL language, a

high-level language with special provisions for expressing vector and matrix computations, with the Honeywell HDC-701P aerospace computer. The high-level architecture versions of a set of benchmark routines were found to require 19% fewer program bits than carefully coded assembly language versions. A timing analysis showed that the high-level architecture programs could be expected to require 14% less execution time.

Tafvelin and Wikström[12] compared a proposed high-level language architecture for the machine oriented high-level language Mary with IBM S/360. A set of seven programs was used, with a total S0360 volume of 42000 bits. The main result is that program size is reduced almost by a factor of 3. This is partially attributable to a sofisticated adressing scheme called "refined display" used in their architecture. No dynamic results are given.

The work by Tanenbaum[5] has already been mentioned. His EM-1 architecture shares several properties with LAX2 but does not have the application orientation of the latter. The performance evaluation he reports is based on a small amount of data. All performance figures concern static code size. Apart from isolated statements and programming constructs he treats only four small programs. Their total size on the PDP-11 is 3776 bits and on EM-1 47% of this figure.

Conclusion

The reported work has given yet another example of the superiority of high-level architecture, designed from language and application considerations, over conventional architecture. The evaluation was partial - the only examined properties were program volume and number of interpreted instruction bits. These quantities were evaluated using a set of complete, realistic programs from a well-known source[3].

The following features contribute significantly to the shown superiority of the high-level architecture:

- efficient subroutine support
- structured memory, short addresses
- application oriented data types and operations.

As stated in the Introduction the goals for the LAX2 design include low cost for software production. The high-level architecture supports this goal by:

- eliminating concepts from low-level programming such as registers, primitive addressing, pointer arithmetic, and goto statements
- easing the compilation process (the basic compiler is available as a machine instruction)
- providing extensive run-time protection.

We are convinced that these properties significantly promote programmer productivity as well as the reliability of the software produced. The continuing rise of the ratio of software cost to hardware cost emphasizes the importance of such "soft" advantages of high-level architecture. Unfortunately they are hard to quantify. To do so for LAX2 would require, in the first place, more practical experience with the machine than is available today.

Acknowledgement

This work was supported by grants from the Swedish Board for Technical Development.

References

1. L-E Thorelli, Description of the high-level machine language LAX2, Part 1, TRITA-CS-7602, Royal Institute of Technology, Stockholm, 1976.

2. G Båge, Description of the high-level machine language LAX2, Part 2, TRITA-CS-7901, Royal Institute of Technology, Stockholm, 1979.

3. B W Kernighan and P J Plauger, Software Tools, Addison-Wesley, Mass., 1976.

4. E A Feustel, On the advantages of tagged architecture, IEEE Trans. on Computers 22(7), 644-656, 1973.

5. A S Tanenbaum, Implications of structured programming for machine architecture, Comm. ACM 21(3), 237-246, 1978.

6. B W Kernighan and D M Ritchie, The C Programming Language, Prenctice-Hall, New Jersey, 1978.

7. The Bell System Technical Journal 57(6:2) (Special issue on the UNIX system), 1978.

8. W T Wilner, Design of the Burroughs B1700, Proc. AFIPS FJCC 1972, AFIPS Press, N.J., 489-497.

9. W E Burr, A H Coleman, W R Smith (Eds.), Final Report of the Computer Family Architecture Selection Committee, Army Electronics Command, Ft. Monmouth, N.J., August 1977.

10. D B Wortman, A study of language directed computer design, Technical Report CSRG-20, Univ. of Toronto, 1972.

11. W C Nielsen, Design of an aerospace computer for direct HOL execution, Proc. ACM-IEEE Symposium on High-Level Language Computer Architecture, New York, ACM, 34-42, 1973.

12. S Tafvelin and Å Wikström, Aspects of compact programs and directly executed languages, BIT 15(2), 203-214, 1975.

Appendix A

ELAX Instruction Summary

Of the 256 available byte values, the ones in the upper half are reserved for producers and locators (byte values in hexadecimal):

- 80-9F: producers, stack variables
- A0-BE: producers, own variables
- C0-DF: locators, stack variables
- E0-FE: locators, own variables.

A producer pushes the value of a variable on the stack. In the case of a composite value, only its head is pushed.

A locator locates the place of a variable and initiates a locator-sequence. The latter is composed as described by the regular expression

locator pursuer* (catcher | effector)

The opcodes used for pursuers, catchers, and effectors are in the interval 00-2F, and the same opcodes are also used for other instructions. This is possible since the same instruction cannot occur both within and outside a locator-sequence.

Pursuers enable remote accessing. The three main pursuers are:

'Pcomp': The located value must be a string (,realarray) or node v. An operand i of type index is required (on the stack). The i'th component of v becomes located.

'Pfirst': The located value must be a string (,realarray) or node v. The first component of v becomes located.

'Pown': The located value must be a prog p. An operand i of type index is required. The i'th own variable of p becomes located.

Catchers push a value on the stack. The three main catchers are 'Ccomp', 'Cfirst', and 'Cown', of the producers above. The value produced is that of a component or an own variable, respectively.

Effectors are categorized as basic effectors, string effectors, and special effectors. The basic effectors are:

'clear': writes the index value 0.

'scratch': writes the value nil.

'store': writes a value popped from the stack.

'plus','minus' (only for index values): adds, resp. subtracts, a value popped from the stack.

'incr','decr' (located value must be index): increments by 1, resp. decrements by 1.

Before summarizing the string effectors some other classes of instructions will be treated.

Constants are instructions pushing a value described by the instruction itself on the stack.

Index constants: Values 0-10 are represented by the byte values 00-0A. Values 11-255 are represented by two-byte instructions. Values 256-16383 are represented by three-byte instructions.

Character constants: Represented by two-byte instructions, where the second byte contains the character code.

The constant nil and the boolean constants true and false have one-byte representations.

String constants: The empty string has a one-byte representation. Other strings have a (n+2)-byte representation, where the first byte is an opcode, the second contains n, and the remaining bytes the character codes of the string (1≤n≤255).

Decimal constants: See ref. (2).

(Real constants: Planned, see ref. (1).)

The remaining data types (see Fig. 1) have no constants.

The instruction class computors contains instructions taking a number of values from the stack and producing a value on the stack. These instructions, like the constants, are side-effect-free. Subclasses of computors include binary operators, unary operators, binary predicates, unary predicates, converters, and creators. All computors have a one-byte representation.

The binary operators are '+', '-', '*', '/', and 'modulo'. They are defined for boolean, index, decimal (and real) operands.

The unary operators are 'negate', defined for boolean, decimal (and real) operands, and 'abs', defined for decimal (and real) operands. ('truncate' and 'round' are planned for reals.)

The binary predicates are 'same', 'diff' for comparison of heads of composite values, and six relational predicates, defined for operands of types boolean, index, decimal (, real), and character and string. - Here, as with most other instructions, a character is regarded as a string of length one.

The unary predicates are 'letter' and 'digit' for character operands, and 'bad', yielding true if and only if its operand is nil, and 'good' - the negation of 'bad'.

Converters convert from one data type to another. In essence, direct conversion is possible between character and index, between index and decimal (, between decimal and real, and between index and real). The converter 'length' produces the length of a string, node, decimal, prog (or realarray).

The creators create a new composite value (head on stack, body on heap). They are 'create', to create a string or node (or realarray) of specified length, 'copy', to produce a copy of a composite value, 'substring', to produce a substring from specified positions in a string, and 'cat', to produce the concatenation of two strings.

The string effectors have the following in common:

- The located value must be an index v.
- At least one operand, a string $s_1 \ldots s_n$, is required.
- v must be less than n.

The effector treats the string segment $s_{v+1} \ldots s_n$ and will normally increase the value of v as a side effect. The aim has been to enable convenient and efficient sequential processing of strings. The string effectors are categorized as predicate effectors, pass effectors, locate effectors,

get effectors, and put effectors. Descriptions of the individual instructions can be found in ref. (1). Here we can only offer an enumeration of them; hopefully their names give some hints of their meanings.

Predicate effectors:
'prefix', 'part', 'subequ'
Pass effectors:
'pass', 'paslet', 'pasdig', 'pasletdig'
Locate effectors:
'locate', 'loclet', 'locdig', 'locletdig'
Get effectors (get value from string):
'getindex', 'getchar', 'getident', 'getdec',
('getreal',) 'getstring'
Put effectors (put value into string):
'putnext', 'putpart'

The next instruction class of interest is the jumps. All jumps are generated from high-level control structures during the TLAX-ELAX compilation. These include, in short:

if - then - else : generates forward jumps
case : generates jump table, an indexed jump, and forward jumps
do - od : generates a backward jump
exits from do-od: generate forward jumps.

In addition, the constrol structure suggested by Zahn (C T Zahn, A control statement for natural top-down structured programming, Programming Symp. Proc. 1974 (Ed: B Robinet), Springer, 170-180) is implemented in LAX2.

All jumps are within progs and relative; distances are coded in one or two bytes. In total 22 opcodes are allocated to jumps.

Additional instructions controlling the flow of computation are:
'exec', 'return': for ordinary procedure (prog) activation.
'init', 'attach', 'detach', 'resume', 'call': used in connection with coroutines (coprogs).
'exit': for abandoning the current computation and reinitialization of the LAX2 process.

LAX2 supports frequency measurements during execution. So-called counters can be placed at arbitrary points in programs; they are (if enabled) automatically incremented each time they are passed during execution. Instructions exist for operating the counters.

Fixprograms are protected programs created at the initialization of a LAX2 process. Some of them are automatically activated by different runtime error events. There are also instructions for activating fixprograms from other programs.

The 'compile' instruction, invoking the TLAX-ELAX compiler , is implemented partially as hidden LAX2 programs. There exist special ELAX instructions only available to these programs, cf ref. (2).

A set of input/output instructions is described in ref. (2).

DIRECTLY INTERPRETABLE LANGUAGE DESIGN FOR HIGH LEVEL LANGUAGE SUPPORT*

B. R. Rau and P. Bose

Coordinated Science Laboratory
University of Illinois
Urbana, Illinois 61801

Abstract

The complexity, in space and in time, of directly interpreting serial, block structured, high level languages is examined. On the basis of this study, it is apparent why it is undesirable to directly interpret high level languages. A systematic procedure is developed for the design of well-matched intermediate languages for supporting high level languages.

1. Introduction

With the steadily increasing emphasis upon the construction of structured, reliable and maintainable software, the trend is toward the use of suitable high level languages (HLLs) in preference to machine or assembly level languages. The computer architect thus, is faced with the task of designing a conducive environment for the execution of HLL programs. This is a shift, in perspective at least, away from the traditional role of the computer architect; no longer is it appropriate to approach the design task at the machine language level.

One viewpoint advocates the direct interpretation of the HLL program, by a interpreter implemented in either hardware, software or firmware, e.g., [1,2,3]. The problems associated with such direct interpretation have been sketched in previous work [4,5] and will be elaborated upon in this paper to demonstrate the general undesirability of this approach. Thus, it will be shown that most HLLs are not directly interpretable by the space-time criteria that are developed subsequently.

The alternative is to translate the HLL program into an intermediate representation that is directly interpretable. Such an intermediate language is termed a directly interpretable language (DIL) [5]. Currently, the DIL most frequently used is the machine language of an available computer. Unfortunately, all too often, the machine language has not been designed with the given HLL in mind leading to significant inefficiencies in time and space. It is of interest, therefore, to understand and formalize the design of a DIL that is well matched to a given HLL and the relationship between the two. Such a DIL could then either constitute the instruction set architecture of a machine dedicated to that HLL, or could be interpreted by a universal host machine (UHM), i.e., a machine which can interpret any DIL with equal and relatively little difficulty. This paper presents some preliminary results relating to the properties of HLLs that disqualify them from being DILs, the relationship between well-matched HLLs and DILs and the process of designing a DIL for a given HLL. Identifying the essential characteristics of the universe of DILs clearly is valuable in determining the architecture of universal host machines.

The primary motivation behind the search for an ideal DIL is the desire to optimize the space-time requirements of the interpretation process. A secondary goal is to facilitate the compilation process. Some interesting space-time measures and analyses of "ideal" intermediate languages have been developed by Hoeval and Flynn [6]. In this paper an attempt is made to approach the design of DILs in a systematic, top-down fashion with no assumptions as to what the end-product should look like. Instead, it is dictated by a systematic methodology that accepts as input a description of the HLL and is guided by current technological limitations.

The DIL design will be effected in this paper by considering the issues and problems involved in directly interpreting a HLL. By removing these problems via a systematic transformation process, the target DLL will be derived. Although no specific host hardware descriptions are considered during the design, such a DIL should (by the definition of a DIL [5]) be one for which it is technologically feasible to build a hardwired interpreter. In other words, it should be possible to view the target DIL as a machine language for a hypothetical computer with certain basic, practically feasible data and control structures. Such

*This work was supported by the Joint Services Electronics Program under Contract DAAB-07-72-C-0259.

specific implementation considerations will be discussed in one of the later sections.

2. A Model of Interpretation

In this section, we shall present a conceptual model of the process of (direct) interpretation of a serial HLL. Some of the main features of the interpretive process will then be illustrated in terms of this model and a specific example high level language. Figure 1 presents the syntax and semantics for some of the productions of our example HLL. The syntax is specified in a context free BNF metanotation; the semantics corresponding to each production, are specified in a semi-formal manner. If not originally so, the source context free grammar (CFG) specification is assumed to have been converted to an equivalent ϵ-free form. The algorithmic methods of achieving such a conversion are well known [7] and are not discussed here. The HLL program of Figure 2 will be used as a working example.

Our conceptual model of interpretation draws heavily upon the concepts in Johnston's Contour Model [8] and Knuth's approach to specifying the semantics of programming languages [9]. It consists of four concurrent, interacting processes:

1. Lexical Analyzer: This process is a string to string transducer which converts the input alphanumeric string into a output string of tokens corresponding to lexemes. The function, operation and complexity of this process are relatively well understood and will not be considered further in this paper.

2. Syntactic Analyzer: This phase of interpretation (also known as parsing or recognition) is in essence a string to tree transduction process, where the string of tokens emitted by the lexical analyzer is converted into a (parse) tree using some convenient parsing strategy.

3. Static Semantic Analyzer: This process is the one which operates on the tree being built by the syntax analyzer by associating with each node the relevant semantic information needed to be able to perform the actions called for by the program semantics. Any propagation of attributes (up and down the tree) required to be performed in order to fully specify the attributes (and hence, the semantic actions) of each node, has to be carried out by this analyzer [8]. Nodes or subtrees deemed useless (i.e. after all relevant attributes have been made use of or transmitted to the root of the subtree) are discarded as the analysis proceeds. This process does not, itself, perform the actions indicated by the program. It merely gathers the information needed and sets up the next process. All data-independent actions that can be performed by analyzing the source program alone, are in the realm of the static semantic analyzer.

4. Dynamic Semantic Analyzer: This process actually performs the semantics of the program, by executing the semantic actions associated with each node of the tree. Subtrees are discarded as soon as the relevant semantic actions have been executed and the attributes are no longer needed by the static semantic analyzer.

It is important to note that the four processes listed above run in a mutually interlocked manner such that each process gets ahead of the next one in sequence only to the extent necessary for the latter to operate. The controlling process is the dynamic semantic analyzer whose actions are specified by the statements following the label "Dynamic actions" in the definition of the semantics in Figure 1. In performing its function, it must make use of certain attributes, termed S-derived, which are evaluated by the static semantic analyzer. S-derived attributes are defined to be those attributes which can be derived by an analysis of the program text (i.e., input data independent). The derivation of these attributes is specified in Figure 1 in an assertive rather than an imperative manner, i.e., their relationship to other attributes is specified instead of a series of statements the execution of which would assign to them their correct value. The manner in which they are derived is deliberately left unspecified. It is implicitly understood that the dynamic semantic analyzer forces the static semantic analyzer to proceed just far enough that the needed S-derived attributes have been evaluated. The syntax analyzer has a pointer, SYN, into the string of lexemes emitted by the lexical analyzer, that points one lexeme beyond the (minimum) amount of the string that the syntax analyzer must have consumed so as to set up enough of the syntax tree for the static semantic analyzer to perform its function. The syntax tree is necessary since the S-derived attributes are necessarily defined in the context of this tree. Generally, the lexical analyzer's pointer, LEX, into the alphanumeric string will correspond exactly to SYN. Assume the dynamic semantic analyzer is executing the semantics of the node labelled $\langle Block \rangle_1$ in Figure 2. This requires knowledge of the number of declarations in the outermost block. To determine this, the static semantic analyzer requires that all the declarations in the outermost block be parsed. Consequently, LEX will be at the "x" immediately following "integer x;".

The manipulation of SYN and LEX is, by and large, implicit. In the case of loops, conditionals, procedure calls and returns, the dynamic actions explicitly alter LEX (and consequently SYN) by a statement of the form "Parse (u,v)" or "Parse and Process (u,v)" where u identifies a character in the program text by its memory address and v is a non-terminal which serves as the goal for the parser. In the case of procedure calls, the current value of LEX is saved explicitly.

In Figure 1, attributes labelled D-derived are evaluated by the dynamic semantic analyzer. An S-derived attribute is termed COPIED if it is merely the copy of an attribute elsewhere. An attribute is INHERENT if its value is an inherent property of that node. In addition, the type of the attribute (INTEGER, REAL, POINTER, etc.) are specified. Figure 1 clearly demonstrates the complexity of the procedure call and return (see productions 10 and 24). Note also that production 21 requires that the text to be skipped be parsed, even though it will not be executed, just to determine where the ⟨Stmt⟩ or ⟨Simpstmt⟩ ends.

3. Space and Time Requirements for Interpretation

The model of interpretation developed in the previous section may be used to obtain a qualitative understanding of the time and space involved in the direct interpretation of HLLs. Although, in practice, the tree representation would probably be discarded in favor of a more compact representation such as a stack, the space occupied by the tree is related by a factor of proportionality and, so, is a good indicator of the actual space requirements. The advantage of the tree representation lies in its conceptual simplicity which is uncluttered by extraneous implementation issues.

The space requirements are five-fold: (1) the space occupied by the program being interpreted; (2) that occupied by the interpreter; (3) that required to hold the portion of the syntax tree that is currently in existence; (4) the space needed to store the attributes associated with the tree nodes; (5) the space occupied by the parse stack which contains terminals and non-terminals that have been scanned by the syntax analyzer but are yet to be reduced. (This is needed when a bottom-up parsing scheme is used.) The total computation time for the interpreter is the sum of the computation times for the individual processes.

An obvious way of reducing the size of the program being interpreted is to replace the alphanumeric string representation of lexemes by more efficiently encoded bit-strings during a pre-processing step. As a result, the lexical analysis process would be eliminated from the interpreter thereby reducing the interpretation time. On the other hand, no longer would one be interpreting the original HLL directly; instead, a closely related language would be the object of interpretation. In this manner, by identifying the problems associated with the direct interpretation of the original HLL and by modifying the HLL only to the extent absolutely necessary to remove these problems, one obtains a language that is as closely related to the original as possible while possessing the property of being directly interpretable. Pragmatically, a language will be considered to be directly interpretable if, in the context of current technology and cost-functions, it is feasible and desirable to directly interpret the language in comparison to alternative strategies. Thus, the demarcation between languages which are and are not directly interpretable is vague at best and may be expected to change with time.

The space occupied by the interpreter is related to its complexity. The dynamic semantic analyzer is central to the interpreter and can, at best, be made more efficient but cannot be eliminated. As shall be shown subsequently, the static semantic analyzer and the syntactic analyzer can be eliminated by suitably modifying the language.

The space requirements for the syntax tree are best minimized by reducing the amount of the tree that is in existence at any one time. This corresponds to those nodes that have not yet been processed and discarded by the dynamic semantic analyzer. Whereas the objective must be to prevent the syntax analyzer from getting far ahead of the dynamic semantic analyzer (to minimize the size of the tree present), there are factors that will prevent the realization of this goal; there are occasions when the dynamic semantic analyzer, to perform its function, requires information (attributes) that the static semantic analyzer can provide only by looking ahead in the tree, which in turn requires that the syntax analyzer have proceeded far enough ahead. The language must be altered to remove such situations. These modifications, by reducing the size of the tree, also reduce the total number of attributes that must be stored and, consequently, the amount of space needed for this purpose.

The fifth space requirement depends upon the parsing strategy that is selected (or imposed by the grammar specification). The two broad classes of parsing techniques are the top-down and the bottom-up methods. Most parsing strategies can be viewed as either one or the other or a hybrid. With the top-down technique, the production to be used is known when the syntax analyzer's pointer into the string corresponds to the left most terminal of that production (with an optional look ahead of k). The input tokens, therefore, may be consumed and acted upon as they are encountered since their syntactic significance is defined when they are first encountered. In contrast, bottom-up techniques know which reduction is to be applied only when the syntax analyzer's pointer is at the token which corresponds to the right most terminal of the corresponding production (once again, with an optional look ahead of k). In general, there will exist a number of terminals (and non-terminals) whose syntactic significance has not yet been established (since the corresponding right handles have not yet been encountered), but which have been already scanned by the syntax analyzer. Space is needed to store these items, generally in the form of a stack. From this point of view, a grammar suited to top-down parsing is indicated.

With respect to interpretation time, there is little that can be done to minimize the time required by the dynamic semantic analyzer beyond

eliminating inefficiencies since the algorithm embedded in the program must be executed. The amount of computation performed by the static semantic analyzer is reduced if the type of attribute propagation can be matched to the parsing strategy. Inherited (synthesized) attributes can be handled easily with a top-down (bottom-up) strategy. However, since both types of attributes are generally involved, the best approach is to explicitly provide certain crucial attributes in the string, thereby implying a further modification to the language.

Before discussing ways of reducing the time expended in syntax analysis, it is instructive to catalog the various reasons for the existence of syntax with a view to totally eliminating the syntax analyzer if possible.

1. Reliability. The major function of syntax at this point is to restrict the user to a set of strings that are meaningful to the language processor.
2. Readability.
3. To remove static semantic ambiguities. The procedure for deriving attributes is defined in the context of the syntax tree which must, therefore, be derived.
4. To remove dynamic semantic ambiguities. Often the dynamic semantics of certain constructs are defined by the syntax tree, e.g., precedence relationships binding operands to operators.
5. To permit an efficient parsing strategy.

In the case of a HLL, all of these points are important and the syntax cannot be ignored; nor can the syntax analysis be eliminated. If the emphasis is placed on the last issue, that of an efficient parsing strategy to reduce the interpretation time, then it may be necessary, as we shall see, to sacrifice some readability. We shall do so to obtain a "high-ish level" DIL.

On the other hand, if we are interested in a related "low level" DIL, i.e., one which is compiled into and then interpreted but never directly programmed in, then only issues 3 through 5 are relevant. Readability is clearly unimportant and reliability is guaranteed since the compiler will not pass any illegal programs. If we further perturb the language so that the semantics are defined independently of the syntax, then syntax analysis is rendered useless and may be discarded altogether. The interpreter may now recognize a degenerate grammar (one with very few productions) which essentially permits any string of terminals. The syntax analysis for such a grammar consists merely of checking for illegal terminals.

Both the high-ish level DIL and the low level DIL are closely related to the original HLL by virtue of the systematic transformations that are listed in the next section. The former DIL may be viewed as a substitute for the HLL if a directly interpretable HLL is deemed essential. The latter DIL is best viewed as a well matched intermediate language for the HLL It is clear that a number of DILs may be defined that are intermediate between these two DILs.

4. A Design Methodology for Directly Interpretable Languages

In the context of the previous discussion, the following sequence of modifications (on the high level language) may be used to arrive at a directly interpretable language:-

(a) Distinct syntactic tokens or left handles (represented by underscored integers in this paper, e.g. $\underline{1}$,$\underline{3}$) are inserted to all production right-hand sides (Figure 3). This makes the grammar LL(1), thus simplifying the top-down syntax analysis phase.

In practice, all productions would not have distinct left handles; only the productions corresponding to the same non-terminal need have distinct left handles. This would drastically reduce the number of syntactic token needed to six. However, in the interests of clarity, we shall retain this redundancy. No changes to the semantics are called for as a result of this step.

(b) Each production right hand side is use-ordered, in accordance with the sequence of semantic specifications attached to that production, i.e., the terminals and non-terminals are placed in the same order in which they are used. Figure 4 shows the productions affected by this step.

(c) Semantic tokens (integers with overscores: $\bar{}$, $\wedge$ or $\vee$) are introduced at selected points in the productions to indicate the need for semantic actions. Of these, the first type of tokens (e.g, $\bar{5}$) calls for semantic action(s) which can be performed without reference to a propagated attribute. Such tokens can thus be scanned and immediately acted upon. The second type (e.g. $\overset{\wedge}{3}$)) references an attribute that is propagated from a node which is to the right in the tree (right-to-left attribute propagation), while the third type (e.g. $\overset{\vee}{6}$) use an attribute obtained from the left (left-to-right attribute propagation). Figure 5 illustrates the effect of applying this step to the selected productions.

(d) The second and third types of semantic tokens (marked $\wedge$ and $\vee$) are replaced, in each case, by a token of the first kind (marked $\bar{}$) followed by an explicit attribute, (e.g. ⟨numb⟩), thereby eliminating the need to propagate attributes at interpretation time. In the last two steps a number of redundant semantic tokens have been defined to enhance clarity. In practice, this redundancy would be eliminated.

(e) All the original terminal symbols (e.g. begin, end etc.) are deleted from the language and the grammar. These symbols, it may be noted, are totally redundant at this point, both syntactically and semantically.

The final form of the DIL grammar at the end of steps (a) through (e) is shown in Figure 6.

It is to be noted, in summary, that our newly derived language (DIL) has the following desirable properties:

1) Top down LL(1) parsing (with no back track) is possible. Thus syntax analysis

is simple.

2) Close tracking between the three interpretation subprocesses is possible, resulting in minimum tree storage requirements and overall speedup in the semantic analysis phase.
3) Due to the closely matched HLL and DIL grammars, a simple syntax-directed translation scheme (SDTS) [10] may be adopted for the translation phase.

It is to be noted, that minimizing the space requirement for holding the DIL program, has not really been considered in listing the modification steps. However, one might guess that the price paid (in terms of increased program size) for achieving the advantages listed above is acceptable.

The language that we have just derived may be used as a high level language in which programming may be performed if the lexemes are represented alphanumerically and the tokens are represented by keywords. This will require the reintroduction of the lexical analyzer. The most unacceptable feature of this language lies in having to explicitly specify the number of lexemes that have to be branched over. The use of labels, while making the language marginally acceptable, would require the equivalent of a one-and-a-half pass assembly phase. The language would no longer be directly interpretable.

If we desire a language that is to be used merely to be compiled into and then directly interpreted, we can continue the transformation process further. Since the need for attribute propagation by the static semantic analyzer is no longer present, syntax analysis at this point is needed only for checking the syntactic correctness of the program. If the DIL is not to be used for direct programming, syntactic checking is unnecessary, since any errors would have been detected during the translation phase. Adopting this point of view, we may proceed to delete all tokens which are purely syntactic (i.e., tokens that are only underscored) from the DIL grammar of Figure 6. The result, now truly resembles an "assembly" language, in that the program consists of a sequence of semantic tokens, or "op codes". Figure 8 shows the program with numerical tokens replaced by alphabetic mnemonics. The simplest grammar that will accept programs in this "assembly" language is the trivial grammar shown in Figure 7, since, the absence of syntax checking implies that any sequence of semantic tokens is acceptable to the interpreter, even if semantically meaningless. If the interpreter is based on this grammar, the syntax analysis process becomes degenerate. The grammar of Figure 6 (after deleting purely syntactic tokens) is needed, nevertheless, to permit the translation of the HLL program into the "assembly" language in a syntax-directed way.

In actual practice, some minimum amount of syntax checking may be desirable even at the "assembly" language level, in which case, the grammar specification would be intermediate between the two "extremes" of Figure 6 (full syntax checking capability) and Figure 7 (no syntax checking).

5. Technological Constraints and Implications

Various assumptions regarding the available hardware and software technology have been implicit up to this point. These assumptions will now be discussed. Firstly, it is assumed that the best technique for the construction of a parse tree is through the use of a pushdown automata. (Compiler theory offers no better alternative). Hence, syntax analysis will necessarily be time-consuming unless the grammar is LL(1).

It is assumed that the large scale use of associative memory will not be cost-effective or acceptable. Hence, information must be represented by data structures that support searching. For instance, the association of an identifier reference to the corresponding declaration (to obtain attributes) would clearly be facilitated by the use of associative memory. In the absence of associative memory, this information must be maintained in data structures (hash tables, linear lists, etc.) which simplify the search. Furthermore, since such searches are, at best, relatively slow, it is preferable to provide explicit attributes in the program which convert the associative search to a well-defined look-up procedure. In the previous example, the identifier reference should be replaced by two attributes consisting of the specification (relative to the current contour) of the contour containing the variable and the ordinal number of the identifier declaration amongst the set of declaration attributes attached to the corresponding Block) node (i.e., an address couple).

Also, it is not evident how a tree structure may be implemented in hardware whereas stacks are readily implementable either in hardware or in software. Thus, whereever possible, tree structures must be replaced by stacks. The sub-tree corresponding to <exp> can be supported by an evaluation stack. If this is done, the semantics associated with certain productions in the grammar must be altered and be expressed in terms of stack operations. If the block retention rules of the language permit (as is the case in our example language), the contour nodes may be maintained on a contour stack and the associated declaration attributes may be allocated space on an allocation stack. As in the Burroughs' B6500 [11], the three stacks may be combined (with a slight attendent increase in complexity).

6. Discussion

The undesirability, in space and time, of directly interpreting most HLLs stems from the need to do syntax and static semantic analyses. Various factors contribute to this need and it has been shown how they can be eliminated to yield a directly interpretable language. The DIL

that is obtained is not unique; two DILs, a low-level one and another higher level one, are derived in this paper by a systematic transformation process. Other trade-offs, not discussed in this paper, exist between the size of the DIL program, the size of the syntax tree and the interpretation time. Thus, a space of DILs exist for each HLL, and the one selected must be specified by further constraints and criteria. Also, precise measures of space and time need to be developed to place the qualitative discussions on a quantitative footing.

Most compilers have a code-optimization phase which performs two functions: machine-independent optimization and machine-dependent optimization. The former consists of program transformations which involve knowledge of the DIL being compiled into. Such optimization is generally self-defeating in a HLL interpreter since the cost of repeated optimization out weighs the benefits accrued. When designing a DIL for a HLL, the presence of the optimization phase in the compiler should not be ignored since it can alter the structure of the syntax tree into a directed acyclic graph (e.g., a common sub-expression's tree may be a sub-tree for a number of nodes). The stack, by itself, may not be an adequate vehicle for implementing such networks. Machine-dependent optimization is present primarily to bridge the mismatch between the semantics of the HLL and the machine language. However, if the "machine" language is designed to match the HLL, this form of optimization may prove unnecessary.

The important issue of encoding strategies for DIL programs has not been touched upon in this paper and, so, program statistics for the HLL have not formed an input to the DIL design process. The encoding technique used can assume various levels of complexity. To begin with, the introduction of redundant syntactic and semantic tokens should be avoided. Assuming that the interpreter will run on a machine that provides for accessing arbitrary length bit-strings (essential for a UHM), the terminals of the DIL should be assigned codes that contain just enough bits to differentiate between the terminals that could have appeared at that point. In this respect, the grammar of Figure 6 is preferable to that in Figure 7 since it reduces the inherent ambiguity at each step. On the other hand, syntactic tokens are now needed and may cause a net increase in program size. Finally, a frequency-based encoding scheme may be employed, defined either on the linear string or on the parse tree [12]. The latter scheme will probably do better, but makes syntax analysis a necessity-yet another space-time trade-off.

The low-level DIL that was obtained is not radical is nature and, in fact, looks quite similar to a number of stack architectures. However, the relationship between features of the DIL and the HLL is now clearer. Also, issues such as the instruction formats to be used, which generally assume a central position in instruction set design, fall out in a natural manner as a result of encoding decisions and conform to, rather than constrain, the other syntactic and semantic requirements of the DIL.

In conclusion, we do not advocate the direct interpretation of sophisticated high level languages since there are far too many costly computations involved that are best factored out and performed just once during a compilation phase. Instead, a well-matched directly interpretable language should be designed along the lines suggested in this paper. Thereby, space-time savings will be achieved and the compilation process will be facilitated.

References

1. K. J. Thurber and J. W. Myna, "System design of a cellular APL computer," IEEE Trans. Comp., C-19, 4, 1970, 291-303.

2. J. P. Anderson, "A computer for direct execution of algorithmic languages," Proc. EJCC, 1961, 184-193.

3. H. M. Bloom, "Conceptual design of a direct high-level language processor," High-Level Language Computer Architecture, Y. Chu (Ed.), Academic Press, 1975, 187-242.

4. L. W. Hoevel, " 'Ideal' directly executed languages: an analytical argument for emulation," IEEE Trans. Comp., C-23, 8, 1974, 759-767.

5. B. R. Rau, "Levels of representation of programs and the architecture of universal host machines," Proc. 11th Ann. Wkshp. on Microprog., 1978, 67-79.

6. L. W. Hoevel and M. J. Flynn, "The structure of directly executed languages: a new theory of interpretive system design," Digital Systems Lab. Tech. Rep. No. 130, Stanford Univ., March 1977.

7. J. E. Hopcroft and J. E. Ullman, Introduction to Automata Theory, Languages and Computation, Addison-Wesley, 1979.

8. J. B. Johnston, "The Contour Model of Block Structured Processes," SIGPLAN Notices, Vol. 6, Feb 1971, 52-82.

9. D. E. Knuth, "Semantics of context-free languages," Math. Sys. Theory, 2, 2, 1968, 127-145.

10. P. M. Lewis, D. J. Rosenkrantz and R. E. Stearns, Compiler Design Theory, Addison-Wesley, 1978.

11. E. A. Hauck and B. A. Dent, "Burroughs' B6500/B7500 stack mechanism," Proc. SJCC, 1968, 245-251.

12. R. E. Sweet, Empirical Estimates of Program Entropy, Ph.D. Dissertation, Dept of Computer Science, Stanford Univ., 1976.

```
1.  <Pgm>          ::= <Block>.
                       Dynamic Actions:-
                       PROCESS (Block);
                       TERMINATE;
                       RETURN;
2.  <Block>        ::= begin<Stmtlist>end
3.                 ::= begin<Declpart><Stmtlist>end
                       N: (S-DERIVED, COPIED, INTEGER) =<Declpart>.N;
                       I: (INHERENT, INTEGER) = -1;   (*  ORDINAL  NUMBER  OF
                       DECLARATIONS *)
                       STATLINK: (D-DERIVED, ^TREENODE);
                       CONTOUR: (D-DERIVED, ^LOCAL);
                       (* TYPE
                             INT-OR-REAL = (^ITTYP, REALTYP);
                             VALUES = RECORD
                                 CASE MODE:INT-OR-REAL OF
                                     INTTYP: (INTVAL: INTEGER);
                                     REALTYP: (REALVAL: REAL)
                                 END
                       ITEM = PACKED RECORD
                                IDNAME: STRING;
                                IDTYP: (INTTYP,REALTYP,PROCTYP);
                                    CASE IDTYP OF
                                       INTTYP,REALTYP: (VAL: VALUES);
                                       PROCTYP:          (ADDR:          INTEGER;
                                           PARMDESCR: ^ATTRLIST;      NUMBPARM:
                                       INTEGER);
                                    END;
                                LOCAL = ARRAY [0..N-1] OF ITEM; *)
                       Dynamic Actions:-
                       NEW   (CONTOUR); (* Create new block environment and
                       associated table (LOCAL) with space for N entries: the
                       pointer  variable  CONTOUR is made to point to this
                       allocated space.   *)
                       ESTABLISH-STATIC-LINK (STATLINK);  (*  This semantic routine
                       assigns the proper value to the pointer variable STATLINK;
                       STATLINK  is made to point to the most immediately
                       enclosing (procedure)-block node if one exists.- otherwise
                       it is set to NIL.    *)
                       PROCESS (Declpart);
                       PROCESS (Stmtlist);
                       CONTOUR  :=  NIL;  (* Deallocation  of  space  and
                       environment   *)
                       RETURN;
4.  <Declpart>     ::= <Decl>;
5.                 ::= <Decl>;<Declpart>
                       N: (S-DERIVED, INTEGER) = 1 + <Declpart>.N;
                       I: (S-DERIVED, INTEGER) = <Father>.I + 1;
                       Dynamic Actions:-
                       PROCESS (Decl);
                       PROCESS (Declpart);
                       RETURN;
6.  <Decl>         ::= integer<Id>
7.                 ::= real<Id>
8.                 ::= procedure<Id>;<Block>
9.  <Decl>         ::= procedure (Id><Frmprmdcl>;<Block>
                       NAME: (S-DERIVED, COPIED, STRING) = <Id>.NAME;
                       I: (S-DERIVED, COPIED, INTEGER) = <Father>.I;
                       ENTRYADDR:  (S-DERIVED,  INTEGER)  = ^<Block>; (* POINTER
                       INTO THE INPUT CHARACTER STRING  *)
                       N: (S-DERIVED, COPIED, INTEGER) = <Frmprmdcl>.N;
                       PARMTEMPLATE:    (S-DERIVED,    COPIED,    ATTRLIST)    =
                       <Frmprmdcl>.PARMTEMPLATE;
                       CONTOUR: (D-DERIVED, ^LOCAL);
                                                                  (*  TYPE ATTRIBUTES = RECORD
                                                                            IDNAME: STRING;
                                                                            IDTYP1: (INTTYP,REALTYP);
                                                                            IDTYP2: (VALTYP,RESTYP);
                                                                            END;
                                                                      ATTRLIST = LIST OF ATTRIBUTES;   *)
                                                                  Dynamic Actions:-
                                                                  CONTOUR := <Ancestor Block Node>.CONTOUR;
                                                                  WITH CONTOUR^[I] DO BEGIN
                                                                      IDNAME := NAME;
                                                                      IDTYP := PROCTYP;
                                                                      ADDR := ENTRYADDR;
                                                                      PARMDESCR^ := PARMTEMPLATE;
                                                                      NUMBPARM := N
                                                                      END;
                                                                  PROCESS (Frmprmdcl);
                                                                  RETURN;
                                                          10. <Frmprmdcl>   ::= (<Frmprmlst>)
                                                                  N: (S-DERIVED, COPIED, INTEGER) = <Frmprmlst>.N;
                                                                  PARMTEMPLATE:    (S-DERIVED,    COPIED,    ATTRLIST)    =
                                                                  <Frmprmlst>.PARMTEMPLATE;
                                                                  Dynamic Actions:-
                                                                  PROCESS (Frmprmlst);
                                                                  RETURN;
                                                          11. <Frmprmlst>   ::= <Pdecl>
                                                          12.               ::= <Pdecl>;<Frmprmlst>
                                                                  N: (S-DERIVED, INTEGER) = 1 + <Frmprmlst>.N;
                                                                  PARMTEMPLATE:    (S-DERIVED,    COPIED,    ATTRLIST)    =
                                                                  <Pdecl>.PARMTEMPLATE || <Frmprmlst>.PARMTEMPLATE;
                                                                  (* '||' is a concatenation operator which takes lists as
                                                                  arguments.  *)
                                                                  Dynamic Actions:-
                                                                  PROCESS (Pdecl);
                                                                  PROCESS (Frmprmlst);
                                                                  RETURN;
                                                          13. <Pdecl>       ::= <Vr>integer<Id>
                                                          14. <Pdecl>       ::= <Vr>real<Id>
                                                                  NAME: (S-DERIVED, COPIED, STRING) = <Id>.NAME;
                                                                  PARMTYP: (S-DERIVED, COPIED, VAL-OR-LOC) = <Vr>.PARMTYP;
                                                                  (* TYPE VAL-OR-LOC = (VALTYP, LOCTYP); *)
                                                                  PARMTEMPLATE: (S-DERIVED, ATTRLIST) =
                                                                                  ( IDNAME = NAME;
                                                                                    IDTYP1 = REALTYP;
                                                                                    IDTYP2 = PARMTYP)
                                                                  Dynamic Actions:-
                                                                  PROCESS (Vr);
                                                                  RETURN;
                                                          15. <Vr>          ::= value
                                                          16. <Vr>          ::= result
                                                                  PARMTYP: (INHERENT, VAL-OR-LOC) = LOCTYP;
                                                          17. <Stmtlist>    ::= <Stmt>
                                                          18.               ::= <Stmt>;<Stmtlist>
                                                          19. <Stmt>        ::= <Simpstmt>
                                                          20.               ::= if<Cond>then
                                                          21.               ::= if<Cond>then else<Simpstmt>
                                                                  CONDVAL: (D-DERIVED, BOOLEAN);
                                                                  Dynamic Actions:-
                                                                  PROCESS (Cond);
                                                                  CONDVAL := <Cond>.CONDVAL;
                                                                  IF CONDVAL THEN BEGIN
                                                                       PROCESS (Stmt);
                                                                       PARSE-AND-SKIP (Simpstmt)
                                                                       END
                                                                  ELSE BEGIN
```

Figure 1. HLL syntax and semantics.

```
                    PARSE-AND-SKIP (Stmt>;
                    PROCESS (Simpstmt>
                    END;
                RETURN;
22. <Simpstmt>  ::= (Id):=(Exp>
                NAME: (S-DERIVED, COPIED, STRING) = (Id>.NAME;
                OBJECT: (INHERENT, VAL-OR-LOC) = VALTYP;
                MODE: (D-DERIVED, INT-OR-REAL);
                VAL: (D-DERIVED, VALUES);
                IDLOC: (D-DERIVED, ^ITEM);
                Dynamic Actions:-
                GETIDLOC (NAME,IDLOC);
                (* This semantic routine follows the chain of static-link
                pointers in searching through LOCAL tables, using the key:
                NAME.  It returns IDLOC, which is a pointer to the ITEM
                obtained on a successful search.  The routine calls
                appopriate error routine(s) if the search is
                unsuccessful.  *)
                IF IDLOC^.IDTYP = PROCTYP THEN ERROR * ASSIGNMENT TO
                PROCEDURE *)
                ELSE BEGIN
                   MODE := IDLOC^.IDTYP;
                   PROCESS (Exp>;
                   VAL := (Exp>.VAL;
                   IDLOC^.VAL := VAL;
                   END;
                RETURN;
23. <Simpstmt>  ::= (Id>
                ::= (Id><Actparmspec>
                NAME: (S-DERIVED, COPIED, STRING) = (Id>.NAME;
                IDLOC: (D-DERIVED, ^ITEM);
                STATLINK, ENVIRON: (D-DERIVED, INTEGER);
                SAVED-LEX, SAVED-SYN: (D-DERIVED, INTEGER);
                N: (D-DERIVED, INTEGER);
                ACTUAL: (D-DERIVED, ^PLOCAL);
                (* TYPE PLOCAL = ARRAY [0..N-1] OF PACKED RECORD
                            IDDESCR: ITEM;
                            PARMTYP: VAL-OR-LOC;
                            CASE PARMTYP OF
                               VALTYP: ();
                               RESTYP: (RESULTLOC: ^ITEM)
                            END;
                Dynamic Actions:-
                GETPROCIDLOC (NAME, IDLOC, ENVIRON);
                (* Similar to GETIDLOC except that in this case the
                routine returns an extra value: ENVIRON, which is a
                pointer to the block node in which the procedure was
                declared.  *)
                N := IDLOC^.NUMPARM;
                NEW (ACTUAL); (* Create new (parameter) block environment
                and associated table (PLOCAL) with space for N
                entries.  *)
                STATLINK := ENVIRON;
                PROCESS (Actparmspec>;
                SAVED-LEX := LEX;
                SAVED-SYN := SYN;
                PARSE-AND-PROCESS (IDLOC^.LOC, (Block>);
                LEX := SAVED-LEX;
                SYN := SAVED-SYN;
                UPDATE-RESULT-PARAMETERS (ACTUAL);
                (* This routine inspects the table PLOCAL (pointed to by
                ACTUAL) and performs the task of updating all (actual)
                result parameters; any required type conversions are done
                before such updating.  *)
                ACTUAL := NIL;  * Deallocation of space and
                environment.  *)
                RETURN;
25.             ::= while(Cond>do(Stmt>
                CONDVAL: (D-DERIVED, BOOLEAN);
                START: (S-DERIVED, INTEGER) = @(Cond>;
                Dynamic Actions:-
                LOOP: PARSE-AND-PROCESS (START, (Cond>);
                      CONDVAL := (Cond>.CONDVAL;
                      IF CONDVAL THEN BEGIN
                          PROCESS (Stmt>;
                          GOTO LOOP
                          END
                      ELSE PARSE-AND-SKIP (Stmt>;
                      RETURN;
26.             ::= for(Id>:=(Exp>step(Exp>until(Exp>do(Stmt>
27.             ::= (Block>
28. <Actparmspec> ::= ((Actparmlist>)
                J: (INHERENT, INTEGER) = -1;
                ACTUAL: (D-DERIVED, ^PLOCAL);
                Dynamic Actions:-
                ACTUAL := (Father>.ACTUAL;
                PROCESS (Actparmlist>;
                RETURN;
26. <Actparmlist> ::= (Exp>
                ::= (Exp>;(Actparmlist>
                J: (S-DERIVED, INTEGER) = (Father>.J + 1;
                ACTUAL: (D-DERIVED, ^PLOCAL);
                MODE: (D-DERIVED, INT-OR-REAL);
                OBJECT: (D-DERIVED, VAL-OR-LOC);
                PARMVAL: (D-DERIVED,VALUES);
                PARMLOC: (D-DERIVED, ^ITEM);
                Dynamic Actions:-
                ACTUAL := (Father>.ACTUAL;
                MODE := ACTUAL^[J].ITEM.IDTYP;
                OBJECT := ACTUAL^[J].PARMTYP;
                PROCESS (Exp>;
                IF OBJECT=VALTYP THEN BEGIN
                      PARMVAL := (Exp>.VAL;
                      ACTUAL^[J].ITEM.VAL := PARMVAL;
                      END
                ELSE (* OBJECT=LOCTYP *) BEGIN
                      PARMLOC := (Exp>.LOC;
                      ACTUAL^[J].RESULTLOC := PARMLOC
                      END;
                PROCESS (Actparmlist>;
                RETURN;
31. (Cond>      ::= (Exp><Relop><Exp>
32.             ::= TRUE
33.             ::= FALSE
34. (Relop>     ::= >
35.             ::= <
36.             ::= =
37.             ::= >=
38.             ::= <=
39.             ::= #
                /
40. (Exp>       ::= (Term>
                MODE: (D-DERIVED, INT-OR-REAL);
                OBJECT: (D-DERIVED, VAL-OR-LOC);
                VAL: (D-DERIVED, VALUES);
                LOC: (D-DERIVED, ^ITEM);
                Dynamic Actions:-
                OBJECT := (Father>.OBJECT;
                MODE := (Father>.MODE;
```

Figure 1. HLL syntax and semantics (contd.).

```
                    PROCESS <Term>;
                    IF OBJECT=VALTYP THEN VAL := <Term>.VAL
                    ELSE (* OBJECT=LOCTYP *) LOC := <Term>.LOC;
                    RETURN;
4'.             ::= <Term>'<'<Exp>
                    MODE: (D-DERIVED, INT-OR-REAL);
                    OBJECT: (D-DERIVED, VAL-OR-LOC);
                    VAL: (D-DERIVED, VALUES);
                    Dynamic Actions:-
                    MODE := <Father>.MODE;
                    OBJECT := <Father>OBJECT;
                    IF OBJECT=LOCTYP THEN ERROR (* Formal parameter cannot be
                    a result parameter in this case.  *)
                    ELSE BEGIN
                        PROCESS <Term>;
                        PROCESS <Exp>;
                        VAL := <Term>.VAL + <Exp>.VAL;
                        END;
                    RETURN;
42.             ::= <Term>-<Exp>
43. <Term>      ::= <Fact>
 4.             ::= <Fact>*<Term>
45.             ::= <Fact>/<Term>
46. <Fact>      ::= <Var>
47.             ::= <Const>
48.             ::= (<Exp>)
49. <Var>       ::= <Id>
                    NAME: (S-DERIVED, COPIED, STRING) = <Id>.NAME;
                    MODE: (D-DERIVED, INT-OR-REAL);
                    OBJECT: (D-DERIVED, VAL-OR-LOC);
                    VAL: (D-DERIVED, VALUES);
                    LOC: (D-DERIVED, "ITEM");
                    Dynamic Actions:-
                    MODE := <Father>.MODE;
                    OBJECT := <Father>.OBJECT;
                    IF OBJECT = VALTYP THEN GETIDVAL (NAME, MODE, VAL)
                    (* This routine is similar to GETIDLOC except that in this
                    case it returns the actual value of <Id>; using MODE
                    (which is the type of the declared variable or formal
                    parameter, any required type conversions are also done by
                    the routine; appropriate error routine(s) are called if
                    the referenced <Id> is found to be of type procedure. *)
                    ELSE GETIDLOC (NAME, LOC);
                    RETURN;
50.             ::= -<Id>
51. <Const>     ::= <Numb>
52.             ::= -<Numb>
```

Figure 4. HLL syntax and semantics (contd.).

```
1.  <Pgm>       ::= 1<Block>.
2.  <Block>     ::= 2begin<Stmtlist>end
3.              ::= 3begin<Declpart><Stmtlist>end
      .                 .                                  Semantics: as before.
      .                 .
      .                 .
5'. <Const>     ::= 51<Numb>
52.             ::= 52-<Numb>
```

Figure 3. Step (a):- Introduction of distinct left handles.

```
22. <Simpstat>  ::= 22<Id><Exp>:=
31. <Cond>      ::= 31<Exp><Exp><Relop>
41. <Exp>       ::= 41<Term><Exp>+
42. <Exp>       ::= 42<Term><Exp>-                         Semantics: as before.
44. <Term>      ::= 44<Fact><Term>*
45. <Term>      ::= 45<Fact><Term>/
```

Figure 4. Step(b):- Use-ordering relevant productions.

```
1.  <Pgm>       ::= 1<Block>53.
                    53: TERMINATE;
3.  <Block>     ::= 3begin<Declpart><Statlist>54end
                    3: NEW (CONTOUR); SET-STATIC-LINK (STATLINK);
                    54: CONTOUR := NIL;
9.  <Decl>      ::= 9procedure<Id><Fmprmdcl>;<Block>
                    9: WITH CONTOUR[I] DO BEGIN
                        (* Enter attributes of Ith <Id> into the allocated
                    table: vide Figure 1  *)
                        END;
10. <Fmprmdcl>  ::= 10(<Fmprmlst>)
21. <Stmt>      ::= 21if<Cond>55then<Stmt>56else<Simpstat>
                    55: IF NOT CONDVAL THEN JUMP @<Simpstat>;
                    56: JUMP (* unconditionally *) <Simpstat>@; (* i.e past
                    <Simpstat>  *)
22. <Simpstat>  ::= 22<Id><Exp>57
                    22: (* Get IDLOC and check type of <Id>:  call  ~ ~ir
                    routines if necessary  *)
                    57: IDLOC^.VAL := VAL; (* Actual assignment of value *)
24.             ::= 24<Id><Actparmspec>
                    24: NEW (ACTUAL);
25.             ::= 25while<Cond>58do<Stmt>59
                    58: IF NOT CONDVAL THEN JUMP 59@; (* i.e past 58 *)
                    59: JUMPBACKWARD @<Cond>;
41. <Exp>       ::= 41<Term><Exp>63
                    63: ADD (* values of <Term> and <Exp> *);
49. <Var>       ::= 49<Id>
                    49:  IF  OBJECT = VALTYP THEN RETURNVAL (* return value
                    of <Id> *)
                        ELSE  (* OBJECT  = LOCTYP *) RETURNADDR; (* return
                    address of <Id>  *)
```

Figure 5. Step (c):- Insertion of semantic tokens: selected productions.

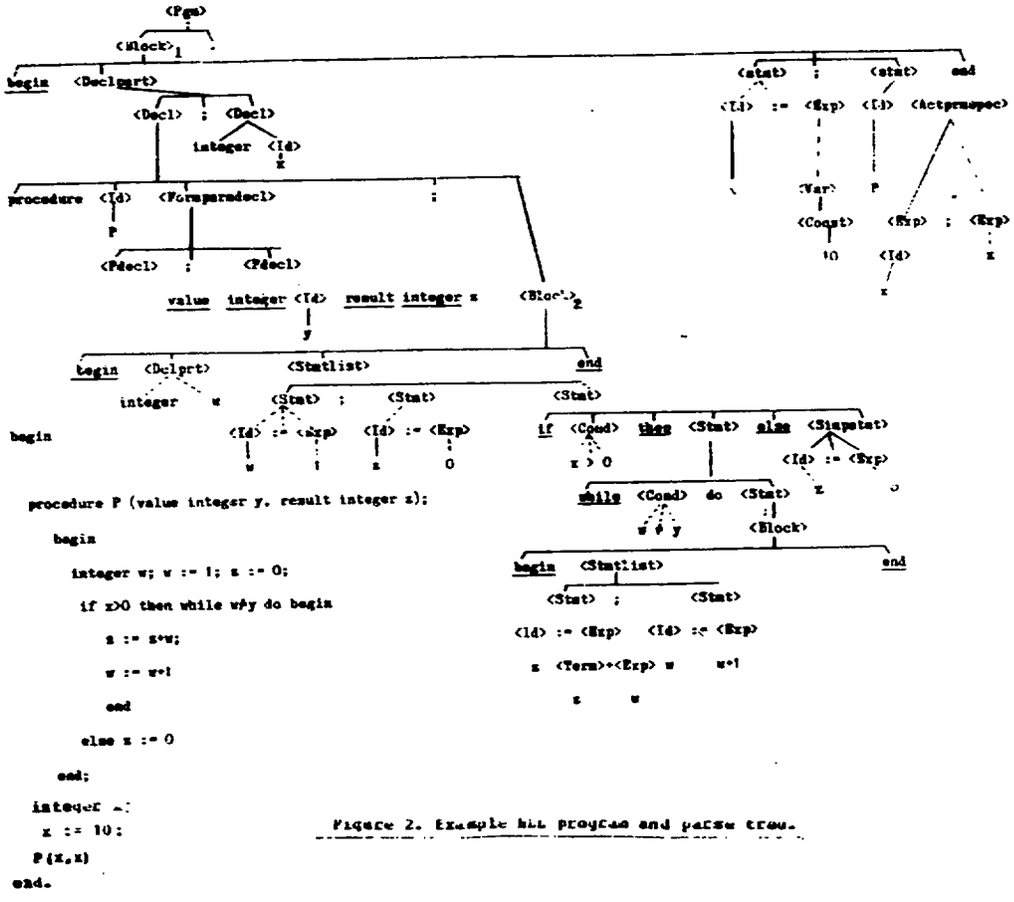


Figure 2. Example HLL program and parse trees.

```
1.  <Pgm>          ::= 1 <Block> 53
2.  <Block>        ::= 2 <Statlist>
3.                 ::= 3 <Numb><Declpart><Statlist> 54
4.  <Declpart>     ::= 4 <Decl>
5.                 ::= 5 <Decl><Declpart>
6.  <Decl>         ::= 6
7.                 ::= 7
8.                 ::= 8 <Numb><Block> 55
9.                 ::= 9 <Numb><Fnparmdcl><Block> 55
10. <Fnparmdcl>    ::= 10 <Fnparmlst>
11. <Fnparmlst>    ::= 11 <Pdecl>
12.                ::= 12 <Pdecl><Fnparmlst>
13. <Pdecl>        ::= 13 <Vr>
14.                ::= 14 <Vr>
15. <Vr>           ::= 15
16.                ::= 16
17. <Statlist>     ::= 17 <Stmt>
18.                ::= 18 <Stmt><Statlist>
19. <Stmt>         ::= 19 <Simpstmt.
20.                ::= 20 <Cond> 56 <Numb><Stmt>
21.                ::= 21 <Cond> 56 <Numb><Stmt> 57 <Numb><Simpstmt>
22. <Simpstmt>     ::= 22 <Exp> 58 <Addrcouple>*
                        58: Assign value of <Exp> to <Id> specified by
                        <Addrcouple>.
23.                ::= 23 <Addrcouple>*
24.                ::= 24 <Addrcouple><Numb><Actparmspec>
                        24: Get location and environment of procedure specified
                          by <Addrcouple>; create new (procedure) block
                          environment with space for N + <Numb> entries:
26.                ::= <Exp> 60 <Exp> 61 <Exp> 62 <Stmt> 63 <Numb>
27.                ::= 27 <Block>
28. <Actparmspec>  ::= 28 <Actparmlst>
29. <Actparmlst>   ::= 29 <Exp> 64 <Val-or-loc>*
                        64: If Val-or-loc = VALTYP then pass value of <Exp> (i.e
                          <Exp>.VAL) as parameter, else pass address
                          (<Exp>.LOC).
30.                ::= 30 <Exp> 64 <Val-or-loc><Actparmlst>
31. <Cond>         ::= 31 <Exp><Exp><Relop>
32.                ::= 32
33.                ::= 33
34. <Relop>        ::= 34
35.                ::= 35
36.                ::= 36
37.                ::= 37
38.                ::= 38
39.                ::= 39
40. <Exp>          ::= 40 <Term>
41.                ::= 41 <Term><Exp> 65
42.                ::= 42 <Term><Exp> 66
43. <Term>         ::= 43 <Fact>
44.                ::= 44 <Fact><Term> 67
45.                ::= 45 <Fact><Term> 68
46. <Fact>         ::= 46 <Var>
47.                ::= 47 <Const>
48.                ::= 48 <Exp>
49. <Var>          ::= 49 <Val-or-loc><Addrcouple>*
                        49: If <Val-or-loc> = VALTYP then return the value of the
                          <Id> specified by <Addrcouple>, else return its
                          address.
50.                ::= 50 <Addrcouple>*
51. <Const>        ::= 51 <Numb>
52.                ::= 52 <Numb>
```

Figure 6. Final DIL grammar after steps (d) and (e).

```
<Pgm>      ::= <Token>
           ::= <Token><Pgm>
<Token>    ::= <Token1>
           ::= <Token2><Numb>
           ::= <Token3><Addrcouple>
           ::= <Token4><Numb><Addrcouple>
<Token1>   ::= INT          5  :(* Enter next entry in contour
                                  table: type integer *)
           ::= REAL         7
           ::= PCR          28
           ::= PUSHTRUE     32 :(* Push a boolean "True" on to
                                  the expression evaluation
                                  stack. *)
           ::= PUSHFALSE    33
           ::= TGT          34 :(* Test if greatGreater Than: Pop
                                  top two elements off the stack: if
                                  the next to the top element is
                                  greater than the top element then
                                  push a boolean "True" on to the
                                  stack, else push "False". *)
           ::= TLT          35
           ::= TEQ          36
           ::= TGE          37
           ::= TLE          38
           ::= TNE          39
           ::= HALT         53
           ::= END          54
           ::= RETURN       55 :(* Return to calling segment:
                                  update result parameters. *)
           ::= INITFOR      60
           ::= STEP         61
           ::= FIE          62
           ::= PASSVAL      64VAL :(* Pass value parameter *)
           ::= PASSADDR     64ADDR :(* Pass address (result
                                  parameter) *)
           ::= ADD          65 :(* Pop top two elements off
                                  stack; add them and push the sum
                                  on to the stack *)
           ::= SUB          66
           ::= MULT         67
           ::= DIV          68
<Token2>   ::= BEGIN        8  :(* NEW (CONTOUR): Create new block
                                  environment with associated with
                                  space for N + <Numb> entries. *)
           ::= PROC         8/9
           ::= PUSHI+       51 :(* Push the number:<Numb> on to
                                  the stack *)
           ::= PUSHI-       52
           ::= BRFC         56 :(* BRanch Forward Conditionally
                                  by amount <Numb> *)
           ::= BRFU         57 :(* BRanch Forward Unconditionally
                                  by amount <Numb> *)
           ::= BRBU         59 :(* BRanch Back Unconditionally by
                                  amount <Numb> *)
           ::=BRBC          58
<Token3>   ::= CALL1        23
           ::= PUSHVAL+     49VAL :(* Push the value of the <Id>
                                  specified by <Addrcouple> on to
                                  the stack. *)
           ::= PUSHADDR     49LOC
           ::= PUSHVAL-     50
           ::= ASSIGN       58 :(* Assign the top of the stack to
                                  the <Id> specified by
                                  <Addrcouple>. *)
<Token4>   ::= CALL2        24 :(* Get the location of procedure
                                  specified by <Addrcouple>: NEW
                                  (ACTUAL): Create new (procedure)
                                  block environment with space for N
                                  + <Numb> entries. *)
```

Figure 7. Final "assembly language" syntax (with partial semantics).

```
BEGIN [2]
      PROC [49]
      BEGIN [1] INT
            PUSHI+ [1] ASSIGN |2,0| PUSHI+ [0] ASSIGN |2,1| PUSHVAL+ |0,1|
            PUSHI+ [0] TGT BRFC [25] PUSHVAL+ |2,0| PUSHVAL+ |1,0| TNE BRFC
            [16]
                PUSHVAL+ |1,1| PUSHVAL+ |2,0| ADD ASSIGN |1,1| PUSHVAL+ |2,0|
                PUSHI+ [1] ADD ASSIGN |2,0| BRBU [23]
                BRFU [4]
      END BRETURN
      INT
      PUSHI+ [10] ASSIGN |0,1| CALL2 |0,0| PUSHVAL+ |0,1| PASSVAL PUSHADDR
      |0,1| PASSADDR
END HALT
```

Figure 8. "Assembly" language program. Numbers in "[ ]" represent literal values; those in "| |" represent address couples. The lexical level of the outermost block (main program) is 0, that of the procedure is 1 and the inner block is at lexical level 2.

Footnotes:- (w.r.t. Fig. 6 *)

* The address couple has the format |lexical level, ordinal number of variable in the declaration list|. For both the numbring starts with 0.

† Val-or-loc is an explicitly propagated attribute which can assume one of two values, specifying, respectively, whether the value or the address of the identifier is required. Since this attribute can assume only two values, it is better taken care of by assuming two different semantic tokens (op codes) where necessary: e.g 49VAL and 49LOC; (vide Figure 7).

TWENTY YEARS OF BURROUGHS HIGH-LEVEL LANGUAGE MACHINES

E. Dean Earnest

Burroughs Corporation
Mission Viejo, California

Abstract

A discussion is presented of several computer systems developments over the past 20 years at Burroughs Corporation. Some of the system design philosophy and concepts employed by the system designers are included to provide an understanding of the motivation of certain design decisions.

Introduction

A discussion of Burrroughs Corporation's 20 years experience with high-level language machines should be considered in the context of some of the concepts and philosophies which served to guide the system designers.

A central theme which has guided the development of computer systems for over 20 years at Burroughs can be characterized as follows:

> The role of computer systems is to facilitate communication between people through the amplification of human capabilities. Anything which creates a distraction from the achievement of this role should be regarded as being wrong.

The use of higher-level languages throughout Burroughs computer systems is consistent with that theme. The development and evolution of efficient machine architectures to support those abstract notations significantly facilitates communication.

The basic set of machine design and use concepts were first publicly discussed by Bob Barton in 1961.[1] The first commercial delivery of a machine whose design was based on this approach (the Burroughs B5000) was made in the early 1960s. The concepts embodied in that system have been expanded over the past 20 years through insights made possible by our accumulated experience in high-level language processing environments.

A brief discussion is presented of some of the concepts and design principles which have guided Burroughs' computer systems design. A review of some representative developments from selected systems design projects is included with some of the design and use ideas which were incorporated.

General Concepts and Ideas

Burroughs' computer systems architecture for the past 20 years is a consequence of the articulation of and adherence to a relatively small set of closely related design concepts and ideas. Following are representative of these tenets:

High-Level Languages

One of the more important concepts introduced with the Burroughs B5000 was a dedication to the use of higher-level programming notation to the practical exclusion of machine or assembly languages. It was proposed and demonstrated that a computer system could be designed and implemented which would provide a sympathetic and efficient host to an exclusively higher-level language processing environment.

At the time of introduction of the B5000, higher-level languages were considered to be of limited practical value in the real world of information processing. Their use consumed vast amounts of resources (particularly time) for the compilation process.

The resource consumption for the compilation process was considered so severe that users frequently abandoned the high-level representation of a program after the initial design and an error-free compilation. They frequently completed the testing and patching process in a more primitive representation. They thereby

avoided solving the basic problem of not having an efficient language processing system. As a result of this multiple representation, the operational program did not resemble the initial high-level description.

In addition to the problems with compilation performance, the object programs executed significantly slower than the proportedly equivalent programs written in lower-level notations. On contemporary machines, both performance observations were valid. The problems confronting compiler writers were significant--conventional machines were not designed to facilitate the mapping of an abstract notation to the set of primitive functions supported by those machines.

In spite of these drawbacks, higher-level languages achieved some acceptance because of the now-recognized advantages of their use for program design, implementation, and enhancement.

Since the B5000 was designed to efficiently handle programs written in ALGOL 60, it was natural to implement all programs, including systems software, in that language.[18] The use of higher-level languages for all programming was critical to the success of the entire project. The approach permitted a continued interaction and feedback among the hardware and software designers, the system implementors, and the system users. During the course of the B5000 project and subsequent developments, the roles of most of the participants in the design changed. Systems designers subsequently became software designers. These, in turn, became software implementors who are included in the population of systems users. The continued, exclusive use of higher-level languages contributes to a fluency in those languages. It also provides strong motivation for the development of an efficient system. At Burroughs, the system users are system designers and are expected to contribute to the hardware and software architectures, implementations, and enhancements.

The viability of using higher-level languages, which was demonstrated on the B5000, reinforced Burroughs' commitment to the approach on subsequent systems designs and program product developments.

It should be noted that while high-level languages have achieved a certain acceptance today, it is largely due to advances in compiler technology. Some modern compilers do achieve an acceptable performance level. Elsewhere in the industry, machines are not being designed to facilitate high-level languages.

The Design Team

A blending of technologies and experience is required for the design of a commercially viable computer system. At Burroughs, a system design team typically consists of a very small group of people from the several necessary disciplines. Each participant must, of course, be well qualified in a particular discipline and must have a good working knowledge in the other represented areas. This cross-discipline knowledge is necessary for effective contribution to the design and implementation decisions.

There has been much written about the integrated hardware/software approach to systems design. Experience has shown that it is not sufficient to collect experienced people from the contributing disciplines. As Bobby Creech observed in his paper on the B6500 architecture, the attitude and the personality of the participants are critical to a successful system design.[2] Intelligence, common sense, and previous experience help considerably, but the successful blending of these three attributes require the correctness of the contributors' attitude and personality.

Design Scope

Bob Barton, as indicated in his 1961 paper on a computer system design approach, suggests that higher-level programming languages should be employed for all programming tasks to the practical exclusion of lower-level notations.[1] Additionally, he believed that the operation of the computer system should be under control of the system itself. This injection of user and operator perspective into the system design process implied a much broader utilization of high-level languages than had been considered in prior systems. Contemporary machines of that era attempted to implement a higher-level language in the hostile environment of a machine/assembly language system. To provide a consistent implementation, the design team on the B5000 broadened their scope of responsibility to include the entire programming and operational environment of the system.

Early in the higher-level language system era at Burroughs, Lloyd Turner and other software team members developed a particularly effective graphical representation of the ALGOL language syntax.[3] This representation significantly clarified the language structure for the team and permitted new insight into an effective compiler implementation. Additionally, this representation and understanding of the language permitted the definition of consistent extensions to the language when other components of systems programming and operation were considered. The entire software system was implemented in ALGOL (as was the ALGOL compiler itself). Since the scope of the systems designers' responsibility encompassed the entire hardware, programming, and operational environment, additional opportunities were available for the partitioning and implementation of required functions. Commonly used functions as well as systems management algorithms were factored out of the users environment into the operating system. Where appropriate, these functions were replaced in the users environ-

ment by calling (naming) syntax which was consistent with the calling language. This system-wide approach to the use of higher-level languages provided a natural environment for the handling of general systems functions. These functions were represented by a syntax which was consistent with that utilized for the systems software. This environment permitted the development and integration of such innovations as automatic memory management, virtual memory and general file management into the operating system. A description of the results of this pioneering effort is included in the B5500 Master Control Program description.[15]

The commitment and the adherence to the exclusive use of higher-level languages throughout the system produced a systems software and usage base which could be readily enhanced. The interface between cooperating software modules implied by the consistent use of higher-level abstractions permits new functions to be easily integrated into the software system. This abstraction also allows software systems to be propagated over several generations of hardware. Software subsystems, such as the Network Definition Language, [4] the Data Management Languages,[5] and augmented operational dialogues which have been implemented over the past several years have been guided by the global perspective suggested by Barton and enhanced by subsequent software teams.

General Design Principles

The preceeding discussion suggests that the recognition of and adherence to a closely interrelated set of sound concepts and design principles provides far-reaching benefits. This conceptual base is required to be successful in the typical commercial systems environment of evolution, growth, and change. In addition to the concepts and ideas previously mentioned, the following are representative complementary design principles which have proven successful at Burroughs.

Recursive Definition. This simple approach can be employed to verify the consistency, completeness, and orderliness of a defined object. Several current notation systems permit solution definition as a recursive process.

Minimal Representation of Information. Not all information has the same importance when considered in a language, program, or system context. The use of a higher-level programming notation wherein information can be represented as appropriate to its static and dynamic usage frequency offers some interesting options to be exploited by system implementors. As an example, Don Knuth has reported on the extremes in FORTRAN function usage in that operational language environment.[17] This representational freedom allows for significant systems performance trade-offs to be effected. Wayne Wilner, in his paper on B1700 memory utilization, presents some interesting observations and comments on the dramatic effects which may be achieved through optimal information representation.[19]

The principle of minimally representing information is consistent with the abstraction of higher-level languages. In natural languages, also, people abstract and codify high-usage communication sequences for efficiency and comprehension.

The Importance of Information Structures. Burroughs' emphasis on the efficient handling of information structures, particularly control structures, has provided far-reaching benefits. The use of the stack in our machine architectures for the partitioning and handling of subroutines, procedures, and processes has permitted the practical application of several of the concepts and ideas noted in this paper. Additional benefits of the use of the stack mechanism include those which contribute to the multiprogramming, multiprocessing, information protection, and control distribution facilities of typical Burroughs systems.

Abbreviated History

Observers of Burroughs systems developments have detected a consistent philosophy regarding systems appearance from the perspective of programmers and users. These observers correctly concluded that the primary impetus for the control and guidance necessary to maintain this image is largely attributable to an informal and long-standing relationship among key Burroughs technical personnel. This group shares both a personal rapport and a commitment to a set of system design and use concepts. In informal meetings and conversations, Barton, Lloyd Turner, and others have served as a catalyst for the elaboration of the original and the synthesis of new ideas and concepts. With this common experience as a basis, it is not surprising that there are repetitions in concept, approach, and appearance within the several Burroughs systems.

Following is a brief discussion, not necessarily in chronological order, of the evolution of some attributes of higher-level language oriented systems at Burroughs. Also included are observations on some of the reasons for particular developments or emphasis.

The B5000, B6000, B7000 Series

In the late 1950s, Burroughs implemented an early version of the ALGOL language on the Burroughs B220, a conventional machine of that era. This implementation served to prove several of Barton's original higher-level language machine concepts. It provided a vehicle for the evaluation, feedback, and refinement of an ALGOL virtual machine.

The B5000 system was announced in 1961. The successor B5500, announced in 1964, included a large, fast secondary storage facility and a more comprehensive operating system. Further enhancements were announced with the B5700 in 1970.

The B6500 system, announced in 1966, incorporated significant enhancements to earlier machines and integrated many new ideas and innovations. The B6800 system, which was announced in 1976, provided a more effective implementation of the B5000 series architecture. It also incorporated features and functions for consistent work and resource sharing among multiple local and/or distributed systems.

The B7700 large-scale system, which was introduced in 1970, provided both source and object-language compatibility with the B6000 series systems. Additionally, it offered enhanced performance, information integrity, and distributed input-output facilities. The B7800, introduced in 1977, is a higher performance version of the B7000 series.

Following are typical of the ideas and concepts of the B5000, B6000, and B7000 systems:

The Stack. Many of the concepts and ideas previously noted were applied in the design of the B5000 system. One of the more important ideas embodied in that machine was the integration of the stack into the machine architecture. The stack mechanism is particularly effective in the ALGOL language handling environment. The power of the stack lies in the control mechanism that can be embedded in it and its use for dynamic temporary storage. This facility permits efficient evaluation of arithmetic expressions and storage of parametric and control information for generalized subroutine and procedure handling. It also allows an effective reduction in program storage requirements since the top of the stack provides an implied address for most of the order codes of the machine. A complete description of the stack and other features of the B5000 and its successor, the B5500, can be found in the Burroughs Reference Manual on those systems.[6,7]

The stack implementation on the B5000 and B5500 was enhanced during the design of the B6500. An evolution of the B6500 stack structure is employed in the current Burroughs B6000 and B7000 series. Based on experience with the B5500, the addressing mechanism for local and global variables was more consistently developed so that the dynamic addressing environment encountered in the execution of programs is maintained automatically by the stack and related structures. In addition, the concept of a "cactus stack" was introduced to provide a vehicle for the more orderly control of multiprogramming and multiprocessing. A good treatment of the use of the cactus stack in process handling is provided by Jack Cleary in his paper on that subject.[8]

The cactus stack may be viewed as a tree of stacks with the trunk containing the basic operating system process representation. Branches from the trunk contain control and parametric information for new processes as they are created. This structure differs from conventional trees in that the trunk can continue to grow after branches have been created. One graphic representation of this structure resembles the Saguaro cactus of the southwest United States--hence the "cactus stack" designation. The paper by Erv Hauck and Ben Dent furnished an excellent discussion of the details of the B6500 stack.[9] Details may be found in the Systems Reference Manual.[16] Elliott Organick's book on the B6700 provides a good treatment of the cactus stack in the context of an overall system description.[10]

The Descriptor. The descriptor on Burroughs' systems is a highly-encoded sequence of program which is executed when it is encountered during accessing of information. The descriptor may be regarded as a generalized form of control word. It is used to separate those functions associated with the information definition and control from procedural code. This separation of description and function facilitates the handling of data and program while maintaining the high-level abstraction of the user environment. Good detailed descriptions of this powerful facility can be found in the paper by Hauck and Dent and in Organick's book on the B6700.[9,10]

The B2000 Series

A major objective of the B2000, B3000, and B4000 systems design was a family of systems which would be efficient at character handling. Specifically, the systems were to provide an effective and efficient host for the COBOL program environment and for character-oriented peripherals such as data communication terminals and magnetic and optically encoded document handlers.

The B2500/B3500 systems were introduced in 1966. The B2700/B3700/B4700 enhancements to the series were announced in 1970 and 1971. The B2800/B3800/B4800 systems which provided both higher performance and machine-language compatibility with earlier systems in the series, were announced in 1975 thru 1977. Many enhancements to the B2000 series have been integrated into the B2900 systems which were announced in 1979.

General Architecture. The experience base for a machine which could perform well in a character-oriented environment began with the B200 systems of the early 1960s and included observations and experience with the B5000 and B5500 systems.[14]

The processor and memory of the B2000-B4000 systems are oriented toward the character, field, and record requirements of the COBOL language. The instruction set accommodates variable-length strings of alphanumeric and numeric representations.

Because of the dominance of field-to-field operations in the COBOL operational environment, the processor was designed to utilize primarily a memory-to-memory instruction implementation. Since the processor retained minimal state between instructions, the system could quickly respond to interrupts from the high frequency of input/output operations in a typical data processing environment. This fast interrupt response facilitated the handling of data communications requirements. It also allowed the handling of the real-time functions of high-volume document handling peripherals in a multiprogramming mix.

The machine also incorporated a stack mechanism to facilitate the handling of control in the COBOL and operating system environments. Since the stack was mapped into the memory area for each program or process, it did not detract from the rapid state-switching requirements of the system.

EDIT Instruction. The application of experience and observations for development and implementation of character handling language and functions is typified by the B2000 series EDIT instruction.

The character handling facilities of the B5000 machine and the necessary primitives to accomplish the COBOL-specified MOVE and EDIT functions were not well designed or implemented on that machine. COBOL was a new programming language at the time of the B5000 design. There was little experience with the practical requirements of that language environment. Additional information was required on the problem of mapping the requirements of the MOVE and EDIT functions on the B5000. The compiler group developed an enumeration and representation of the functional requirements defined by COBOL. They then performed a simulation of the virtual machine implied by that form and semantics. This experience and the resultant insights provided a sufficient basis for the appropriate generators in the COBOL compiler for the B5000. The representation, algorithms, and techniques developed for the B5000 compiler were supplemented by the results of observations on that virtual machine. This experience served as a basis for the design and implementation of the MOVE/EDIT instruction on the B2000, B3000, B4000 systems. On those machines, most MOVE verbs in COBOL can be performed by a single instruction.

Details of the structures and operations implemented on this family of systems can be found in the Reference Manual for those systems.[11]

The B1000 Series

The current Burroughs B1000 series (B1700, B1800, B1900), were designed to support a multiplicity of high-level language and processing environments. In addition, the system was intended to support the emulation of several existing and/or proposed machines.

The initial systems of the B1000 series, the B1700s, were announced in 1972 and 1973. The B1800s, which incorporated significant perfor-

mance enhancements were introduced in 1976. Initial B1900 systems were announced in 1979.

The Design. Based on analysis and experience, the design team concluded that the range of representations and functions dictated by the proposed set of programming languages and machines could not be directly accommodated with a single, commercially viable architecture. A sufficiently small set of structures and operators could not be defined which was efficient for all languages and processing environments. A machine architecture was indicated which could be adapted to each processing and language requirement.

The B1700 system design included an attempt to define a machine which had no inherent structure and no a priori instructions. To satisfy this design objective, a passive machine was required which could accommodate definable information structures and instructions.

The design approach used on the B1700 system was to anticipate a unique machine architecture for each programming language and emulation environment. The designers had to consider both the typical high-level forms of program representation as well as machine-language forms from existing machines. Restated, the B1700 design objective was to efficiently emulate a set of real and virtual machines.

Variable-Field Handling. The ability to vary the machine's image for each emulation environment implies some very specific hardware and software adaptations. Fortunately, our experience on several prior machine designs and research projects suggested several potential solutions to this variable-environment processing problem.

It was observed that data and program are frequently not suited to the representation imposed by typical word or character organized storage and processing elements. The actual nature of program and data demands variable size representation. Considering the range of storage and processing environments of the B1700 system, the smallest unit of information, the bit, must be addressable in order to provide complete flexibility in the mapping and processing solutions. To accommodate this requirement, the B1700 system was designed with a defined-field storage capability. In this memory system, all storage is addressable to the bit, all field lengths are expressable to the bit, and storage hardware fetches and stores one or more bits from any location with equal facility.

The B1700 processor was designed to provide an efficient vehicle for the emulation of multiple language processing environments. The instruction set of the machine included primitives from the set of programming language and emulation environments as well as those which contribute to the emulation, or interpretation, process itself. For example, the Arithmetic-Logic Unit could be parameterized to a width which corresponds to the data or machine being handled. A good exposition of the B1700 design was provided by Wayne Wilner in his paper on that subject and is detailed in the System Reference Manual.[12,13] The book by Organick and Hinds contains an excellent description of the B1700/B1800 systems architecture and application.[20]

Language-Specific Machines. The congruency of the functions dictated by a processing environment and the repertoir of structures and operators supported by a machine generally determines the efficiency of a system. For the B1000 systems, an "ideal" machine was designed for each processing environment. Where an existing machine was to be emulated, the form and semantics of that machine constituted the definition. After the machine definition, an emulator, or interpreter, was developed which provided the semantic definition of that virtual machine. Thus, the compiler writers had an ideal machine structure and operator set for their object code. This repertoir of structures and operators provided an isomorphic relationship between most functions expressed in the high-level language and the target machine.

Optimization. Since the virtual machine could be adapted to each processing and language environment, facilities were integrated into the design to optimize the adaptations. Tools and techniques were indicated which could supplement our perception of the environment with empirical information.

Both hardware and software facilities were integrated into the system to permit static and dynamic observations on the virtual machine's representation and performance. These observations were utilized to extend our knowledge base on these language-specific machines. Virtual machine definition and representation are changed

as indicated by static and dynamic observations on the machine's behavior. This technique, and the adaptability of the machine, has permitted very effective enhancement and optimization efforts to be realized.

It should be noted that the exclusive use of higher-level languages contributes significantly to the success of the optimization efforts. The use of abstract programming notations provides the necessary representational freedom to effect the indicated virtual machine changes. Some additional background material and experience with the application of the systems monitor facility is provided by Russ Hagen in his paper given at a computer performance seminar.[21] A description of the supplemental functions provided in a performance measurement subsystem can be found in the System Performance Monitor Reference Manual.[22]

Resource Management. The B1000 systems support the concept that the machine should manage its own environment. These systems incorporate the standard Burroughs set of operating systems scheduling and other resource management facilities. Program and information segments are handled automatically for both interpreter and virtual machine processes.

At a typical installation, several language environments may be concurrently active in a mix of programs. Through appropriate information integrity and resource management mechanisms, each user views the system as a dedicated facility designed to effectively accommodate his particular language environment.

Summary

The comprehensibility of communications as a result of the exclusive use of higher-level notations throughout Burroughs computer systems enhances their role in human communication. The development and evolution of efficient machine architectures to support abstract information representations makes the use of higher-level languages effective and practical.

Acknowledgement

Many people have contributed to the set of concepts, ideas, and design principles included in this paper. Their application in Burroughs is a tribute to the strong commitment and persistence of Bob Barton and the B5000 team. This group, and the many participants in Burroughs developments over the past 20 years, have expanded and amplified the basic set of ideas.

The author wishes to thank John McClintock and Barbara Bennett for their conscientious criticism of various drafts of this paper.

References

1. A New Approach to the Functional Design of a Digital Computer. R. S. Barton. Western Joint Computer Conference Proceedings (1961). Association for Computing Machinery, New York.

2. Architecture of the B6500. B. A. Creech. Proceedings COINS--69 Third International Symposium. (1969).

3. A Syntactical Chart of ALGOL 60. (1961). W. Taylor, L. Turner, R. Waychoff. Communications of the Association for Computing Machinery. Vol. 4, No. 9.

4. Network Definition Language Manual. Burroughs Corporation, Detroit, Mi.

5. B7000/B6000 Series DMSII DASDL Reference Manual. (1978). Burroughs Corporation, Detroit, Mi.

6. Burroughs B5000 Information Processing Systems Reference Manual. (1964). Burroughs Corporation, Detroit, Mi.

7. Burroughs B5500 Information Processing Systems Reference Manual. (1964). Burroughs Corporation, Detroit, Mi.

8. Process Handling on the Burroughs B6500. J. Cleary. Proceedings of Fourth Australian Computer Conference (1969). The Griffin Press, Adelaide, South Australia.

9. Burroughs B6500/B7500 Stack Mechanism. E. A. Hauck and B. A. Dent. Proceedings 1968 Spring Joint Computer Conference, Thompson Book Company, Inc. Washington, D.C.

10. The B5700/B6700 Series. (1973). Elliott I. Organick. Academic Press, New York.

11. Burroughs B2500/B3500 Systems Reference Manual. Burroughs Corporation, Detroit, Mi.

12. Design of the Burroughs B1700. W. Wilner. 1972 AFIPS Conference Proceedings, Vol. 41, Part 1. AFIPS Press, Montvale, N.J.

13. Burroughs B1700 Systems Reference Manual. (1972). Burroughs Corporation, Detroit, Mi.

14. Burroughs B200 Information Processing Systems Reference Manual. Burroughs Corporation, Detroit, Mi.

15. A Narrative Description of the Burroughs B5500 Disk File Master Control Program. (1966). Burroughs Corporation, Detroit, Mi.

16. Burroughs B6500 Information Processing Systems Reference Manual. (1969). Burroughs Corporation, Detroit, Mi.

17. An Empirical Study of FORTRAN Programs. (1971) Software--Practice and Experience, Vol. 1.

18. Report on the Algorithmic Language ALGOL 60. (1960). Communications of the Association for Computing Machinery. 3, No. 5.

19. Burroughs B1700 memory utilization. Wayne Wilner. 1972 AFIPS Conference Proceedings. Vol. 41, Part 1. AFIPS Press, Montvale, N.J.

20. Architecture and Programming of the B1700/ B1800 Series. (1977). Elliott I. Organick, James A. Hinds. North-Holland. New York.

21. System Performance Indicator (SPI) Monitor System. (1976). Russell L. Hagen. Proceedings of the Burroughs Computer Performance Seminar at U. C. Santa Cruz. Burroughs Corporation, Detroit, Mi.

22. Systems Performance Indicator (SPI) Monitor System Reference Manual (1976) Burroughs Corporation, Detroit, Mi.

A SURVEY OF HIGH-LEVEL LANGUAGE MACHINES IN JAPAN

Masahiro YAMAMOTO

Nippon Electric Co., Ltd., Central Research Laboratories
4-1-1 Miyazaki, Takatsu-ku, Kawasaki 213, Japan

Abstract

Many high-level language machines in Japan have been made which can use most high-level languages. Several proposals and experiments were performed since the late 1960S and significant research started after 1975.

Much of them are proposed on experimental machines. There are a few commercial high-level language machines. It is characteristic that much LISP and APL machine research has been achieved at Laboratories and Universities and a few FORTRAN and COBOL machines have been made by computer manufacturers.

Introduction

This survey report is an overview of the activities related to high-level language machines in Japan. Commercial, experimental and proposed machines are covered. More space is devoted to significant characteristics in their intermediate-language architectures, hardware structures, software/firmware/hardware tradeoffs and evaluation data, rather than their detailed architectures and hardware configurations in order to cover most high-level language machines. For easy understanding and clarification of their differences, architectural comparisons between high-level language machines for the same high-level languages are considered.

High-level language machine research in Japan has been made for most high-level languages. Much of them, however, concentrate on experimental-level high-level language machines and there are only a few commercial-level high-level language machines. Several proposals and experiments were made in the end of 1960S and early 1970S. Significant research efforts have started after some 1975, as shown in Fig. 1.

Generally speaking, it is characteristics that much research data have been gathered on LISP and APL machines at Laboratories and Universities and a few FORTRAN and COBOL machines have been made by computer manufacturers.

References are listed at the end of this report in which the reader can find detailed information. Unfortunately, most of them are written in Japanese.

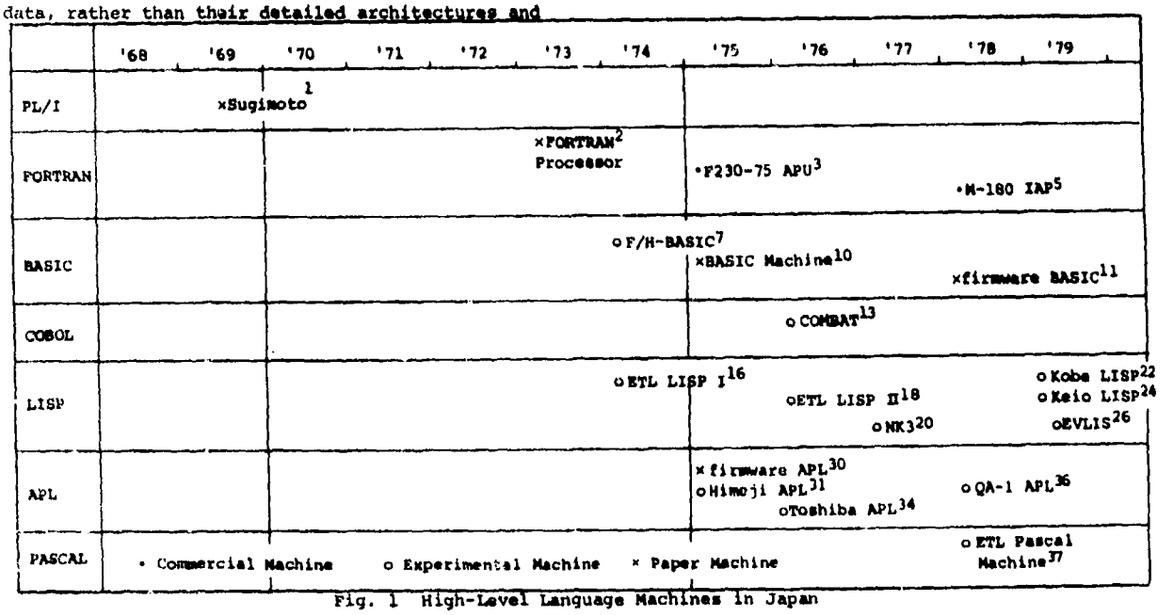


Fig. 1 High-Level Language Machines in Japan

High-Level Language Machines

PL/I Processors

PL/I is the most complex commercial high-level language. Hence it is time-consuming to manipulate on a conventional computer. Therefore, the appearance of advanced and consistent PL/I processors has been desired for quite a while.

The first significant step in the research on high-level language machines in Japan occurred with the proposal for a PL/I processor by M. Sugimoto. In 1969, he proposed a PL/I processor[1] composed of a translator, called the PL/I reducer, and a hardware interpreter, called the direct processor. The PL/I reducer translates a PL/I program into a list-structured intermediate language, DIPL (Direct Processor Input Language), that consists of four parts, Program Structure List (PSL), Statement Normal Form List (SNFL), Attribute List (AL) and Constant List (CL). The direct processor consists of several functionally autonomous units, as shown in Fig. 2.

The PL/I reducer has been implemented. For typical scientific programs, the object code length has been reduced by a factor of 25% on the average, compared to that of the object code generated by the PL/I compiler available at that time. According to the timing simulation program for the direct processor, it was shown that 28% speed gain over the conventional computing system can be obtained for arithmetic/string operations.

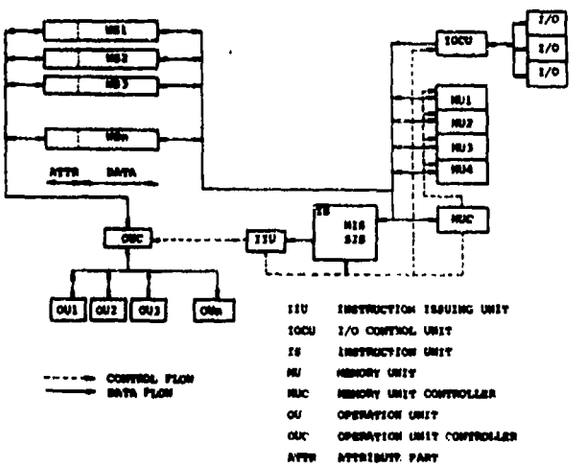


Fig. 2 Block Diagram of the Direct Processor

FORTRAN Processors

FORTRAN or array processors are only used as a commercial high-level language machine in Japan. Some of them have actually been used as an attached or integrated processor in a conventional general purpose computer system for performance enhancement of FORTRAN program execution. Also, in accordance with recent urgent requirements for effective execution of large scale scientific applications, more powerful array processors have been planned.

In 1973, S. Takahashi et al. at Hitachi Ltd. reported results of fundamental, experimental research efforts on a firmware FORTRAN processor[2], where FORTRAN source statements are translated into both reverse polish and mixed reverse polish intermediate texts. In mixed reverse polish, arithmetic statements are translated into reverse polish texts and IF statements are translated into normal polish texts, except for arithmetic expressions in them. The authors concluded that the execution time ratio for reverse polish and mixed reverse polish built in microprograms, reverse polish in software and object machine codes is 0.8 : 1.3 : 9.7 : 1, based on a FORTRAN dynamic statement mix. On the other hand, the object memory capacity ratio is 0.53 : 0.58 : 0.58 : 1, based on a FORTRAN static statement mix.

The FACOM 230-75 APU (Array Processor Unit)[3,4] from Fujitsu Ltd. is a pipelined vector machine attached to a FACOM 230-75 system in which the APU and CPU (Central Processor Unit) share the main memory (Fig. 3). The APU machine structure is characterized by various kinds of internal registers (vector registers, data registers and base registers), vector descriptors and powerful vector instructions for array or vector operations. A FORTRAN user's program is written in AP-FORTRAN which is an extension of standard FORTRAN to include vector functions. It was indicated that the maximum APU performance is 22 Mega Floating-Point-Operations and the APU system performance of various application programs written in AP-FORTRAN is 4-20 times that for corresponding CPU programs. An APU system was installed in Japan's National Aerospace Laboratory.

The IBM System/360 - 2938 AP and the FACOM 230-75 APU are an attached processor to the central processor through an I/O channel or a shared main memory. In order to solve problems, wherein a large amount of hardware was necessary and that a special description using non-standard FORTRAN would be required, Hitachi Ltd. developed the M-180 IAP (Integrated Array Processor)[5] where array processing functions are included within a central processing unit as a general instruction set (vector instructions). A concise vector instruction set, consisting of 28 instructions, was selected based on an analysis of the statistics on the behaviour of FORTRAN programs, obtained using a software tool, FORMAP 8. In the M-180 IAP, FORTRAN user's programs written in standard FORTRAN are vectorized through the vectorizing FORTRAN compiler[6]. It was shown that about 50% of the benchmark programs using execution steps can be vectorized by 28 vector instructions.

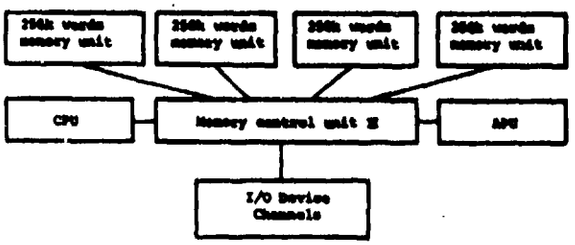


Fig. 3 The FACOM 230-75 APU System Configuration

BASIC Machines

In 1974, Y. Nagai, M. Yamamoto et al. of NEC Ltd. quantitatively analyzed software/firmware/hardware tradeoffs in a BASIC interpreter. For this purpose, three kinds of high-level language machines, a software-implemented BASIC interpreter (S-BASIC), a firmware-implemented interpreter (F-BASIC) and a firmware implemented interpreter with additional hardware (H-BASIC), were implemented. F-BASIC[7] is implemented with firmware on the General Purpose Microprogrammed Simulator (GPMS)[41]. To reinforce the F-BASIC performance, hardware functions, such as transfer/pointer operations, associative functions and so on, were introduced into the H-BASIC[8] on the microinstruction level. Each BASIC processor translates a BASIC program into a same intermediate language, and then interprets it. Experimental results[9] show that 17 times performance improvement is obtained by adopting firmware. 3.6 times more performance improvement was obtained by introducing appropriate hardware functions. The memory capacity necessary for a language processor was also reduced.

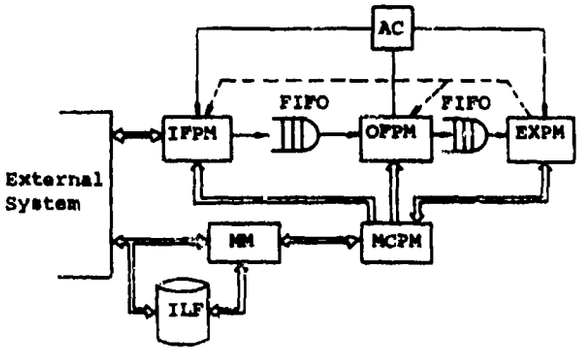
M. Yamamoto, an implementor of the preceding experiment, proposed an advanced high-level language architecture[10] for a BASIC machine as an extenstion of the above three BASIC interpreters in 1975. The BASIC machine is capable of both translation and interpretation of a BASIC program and is characterized by a tagged architecture, a large number of general purpose registers and powerful machine instructions. In addition, bit-handling, masking and table-pointer operations are also installed. It was estimated that the BASIC machine performance is about 2 times that of F-BASIC.

T. Maruyama of Himeji Institute of Technology made a BASIC interpreter[11,12] on a general purpose minicomputer, HP-21MX. Using a software translator, BASIC programs are translated into intermediate languages, which are interpreted by a firmware interpreter. In the interpreter, commonly usable functional routines for such as table pointer/entry manipulations, data conversions and arithmetic operations, rather than for the whole of a special statement, are implemented with microprogram techniques, based on execution frequency evaluation data. The microprogram amount is about 1.3 k words. A firmware BASIC interpreter is about 4 to 9 times faster than a software version on benchmark test programs.

COBOL Machine

COBOL is the most commonly used commercial programming language. It is used for some 70% of all programming. Therefore, hitherto, conventional computers with specialized functions or architecture for COBOL and COBOL machines appeared at the commercial level overseas.

On the other hand, in Japan an experimental COBOL machine[13] similar to NCR COBOL Virtual Machine has been put into implementation since 1975 in NEC Ltd. The COBOL machine architecture, called COMBAT (Cobol Oriented Machine Basic Architecture), has many facilities for efficient COBOL program execution, e.g. many internal data, data descriptors and intensive COBOL function capabilities. The COBOL machine hardware is functionally composed of three processor modules for instruction fetch, operand fetch and instruction execution as shown in Fig. 4. It was indicated that the COBOL machine execution time[14,15] is about 3-5 times faster than that in a medium scale conventional computer. The COBOL machine is running as a processor attached to the conventional commercial computer.



AC: Advance Controller
ILF: Intermediate Language File
FIFO: First In First Out Memory
IFPM: Instruction Fetch Processor Module
OFPM: Operand Fetch Processor Module
EXPM: Instruction Execution Processor Module
MCPM: Memory Control Processor Module
MM: Main Memory

Fig. 4 COBOL Machine Configuration

LISP Machines

The most researched high-level language machine in Japan is a LISP machine. Since the LISP language has many intensive characteristics, e.g. dynamic data allocation, recursive function call and list porcessing, it is impossible to effectively execute LISP programs on conventional computers. Increase in research areas for symbol manipulation and advent of low cost, highly functional and easily usable microprocessors have been accelerating the demand for LISP machines since 1970 in Japan.

An early experiment on a LISP machine was made by T. Shimada et al. of Electotechnical Laboratory (ETL) in 1974. LISP machine research in ETL has been performed in three steps. The first experiment involves a microprogrammed LISP interpreter[16,17] on a user microprogrammable computer, HP-21MX. A Babrow stack model is implemented with microprogram techniques, on which LISP interpreter is made with LISP oriented highly efficient instructions. Also backtracking and coroutine functions are adopted. It was concluded that about 5 to 6 times faster than HP-2100 machine instruction codes is attained. Moreover, much basic evaluation data about microprogrammed LISP interpreter were obtained. It is shown that highly efficient decision making including multi-path jump, recursive call at the microprogram control level, bit manipulation and main memory control are effective for a LISP interpreter.

Based on these evaluation data and experience, new LISP machine (ETL LISP II)[18] was implemented on a universal emulation machine, ACE (Adaptive Computing Element)[42]. Internal data format and interpreter structure for this LISP machine are identical to the HP-21MX version. In order to attain better performance, however, all the interpreter is written in microprograms, and stack configuration, hardware register utilization and memory management are improved due to using advanced ACE hardware facilities.

In addition, virtual LISP machine[19] is being implemented on a powerful 16-bit microcomputer, whose conceptual structure is shown in Fig. 5. In the virtual LISP machine, intermediate language instructions directly corresponding to LISP functions are considered.

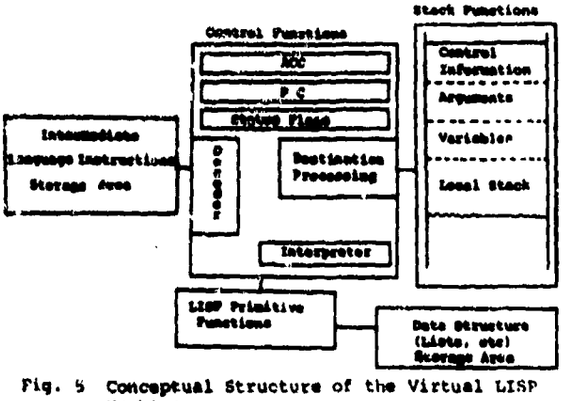


Fig. 5 Conceptual Structure of the Virtual LISP Machine

LISP machine NK3[20,21] of Kyoto University is based on a LISP oriented special processor, which is 32-bit data length, 42-bit microinstruction length and 64-bit list-cell length. Also, it has special hardware units, such as a transfer table for generating microinstruction branch addresses to aid checking for tag field and data category and a hardware stack, whose top areas are always stored in a fast buffer memory. NK3 has about 150 macroinstructions mainly for stack and tag manipulation, in order to effectively execute LISP functions. The processing speed of a LISP interpreter on NK3 is 5-6 times that of a LISP system on a general purpose minicomputer. Figure 6 shows an NK3 blockdiagram.

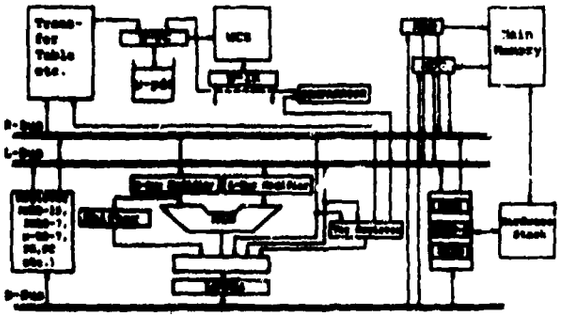


Fig. 6 Block Diagram of LISP Machine NK3

Research on LISP machines in Japan was promoted by the advent of low-cost, high-performance and easily usable microprocessors, specially bit or byte slice microprocessors.

K. Taki et al. at Kobe University developed a LISP processor[22,23], organized with 4-bit slice microprocessors (Am 2900 series), which has 16-bit data length, 56-bit microinstruction length and 32-bit list-cell length. It also has special hardware components characterized by a 16-bit 4-k word hardware stack, a field extractor for data masking and shifting, a 3-bit 1 k word mapping memory generating a 3-bit usage code corresponding to the main memory address and a 1-bit 64 k word bit-table supporting garbage collection function. Figure 7 shows the hardware structure for the Kobe University LISP machine, which is connected to a general purpose computer, FACOM 230-38, through an 8080 microcomputer. A DEC LSI-11 minicomputer performs initiation and maintenance functions, LISP program loading and input/output operations.

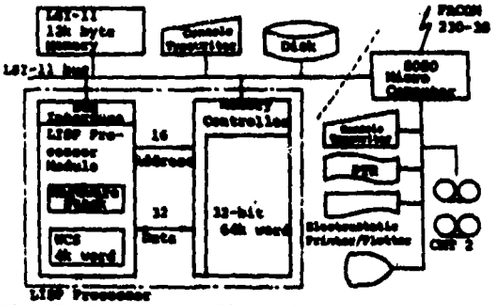


Fig. 7 Hardware Configuration of A LISP Machine System

T. Usuki et al., from Keio University, implemented a LISP machine[24,25] on a multi-microprocessor system, which is composed of an interpreter processor (IP), a storage management processor (SMP) and an input-output processor (IOP). IP performs overall control of LISP program processing and LISP program's interpretation, and has a 16-level hardware stack for sequence control and list manipulation capabilities. Garbage collection and CONS, RPLACE and RPLACD function execution are achieved independently of interpretation on SMP, which is organized of byte-slice microporcessors, 32 special registers and a writable control storage. Garbage collection function is attained based on Dijkstra's algorithm. IOP, a general purpose minicomputer (NOVA), accomplishes input operation of a LISP S-expression, conversion from it to internal forms and file processing. Figure 8 shows the configuration of an experimental multiprocessor system.

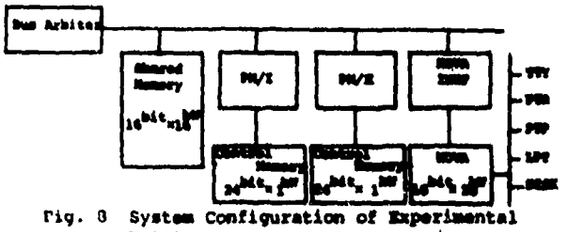


Fig. 8 System Configuration of Experimental Multi Processor System

H. Yasui et al. of Osaka University have been developing a new multiprocessor LISP machine, EVLIS machine[26,27]. In a traditional multiprocessor LISP machine, list processing and garbage collection or I/O processing are performed in a parallel mode. On the other hand, in EVLIS machine, each argument for a LISP function, EVLIS, is parallelly evaluated on multiple processors. It is based on the concept that parallel interpretation of EVLIS arguments is possible if an argument evaluation does not affect the other argument because of its list alteration operation. Figure 9 shows the system configuration of the EVLIS machine, in which an evaluation processor can accomplish an argument interpretation. An evaluation processor is organized of Intel bit-slice microprocessors, I 3000 series, and is 20-bit data length and 50-bit microinstruction length. A 10-bit list cell can be brough into a CAR-CDR register from a main memory. When there is garbage collection function requirement, all evaluation processors stop interpreting EVLIS arguments and parallelly perform their function. A simulation result related to the performance enhancement due to multi processors was shown in the paper[27].

Typical LISP machines have been surveyed. Table 1 shows a summary of their major characteristics. In addition, there are other research efforts related to LISP machines. ALPS/I (Aoyama List Processing System/I)[28] is a compact, low-cost LISP machine on a universal 8-bit microprocessor (i 8080). L. Goto, T. Ida el al., of the Institute of Physical and Chemical Research, are designing a machine for numerical, symbolic and associative computing, FLATS (Fortran and Lisp machine with Associative features for Tuples and Sets)[29]. In FLATS, overflow free and variable precision arithmetic, table look-up computation, and associative computation are realized by hashing hardware, tag mechanism and hardware list processing.

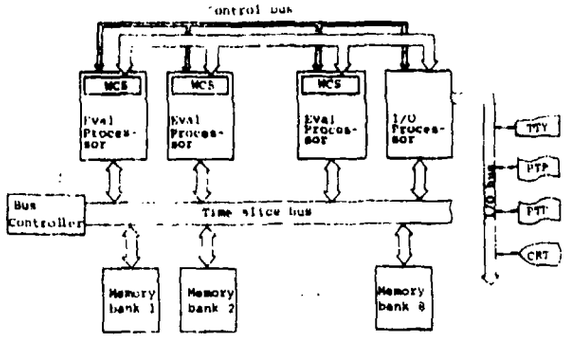


Fig. 9 System Configuration of EVLIS Machine

Table 1 Architectural Comparison Between LISP Machines

	Intermediate Language Architecture	Processor Configuration	Special Processor	Parallel Processing	Hardware Stack	Garbage Collection	Miscellaneous
ETL LISP II	(Direct Interpretation of LISP texts)	ACE + NOVA	Univeral host 16-bit microprocessor	-	Babrow Stack Model Ring Mode Stack (8 words)	-	-
NK3	CAR CDR CONS	Special Processor +Interdata 8/32	Special hardware Processor	-	Fast Buffer Memory	-	Transfer Table
Kobe LISP	CAR CDR CONS	Special Processor +LSI 11	Bit Slice Microprocessor (Am 2900)	-	4k word hardware stack	Bit Table	Mapping Memory
Keio LISP	(Direct Interpretation of LISP texts)	3 Special Processors	Byte Slice Microprocessor	Execution & Garbage Collection	-	-	-
EVLIS Machine	(Direct Interpretation of LISP texts)	4 Special Processors	Bit Slice Microprocessor (I 3000)	Parallel execution of List Processing	-	-	-

APL Interpreters

APL has many features to be implemented by firmware/hardware techniques, some of which are (1) dynamic data and dimension attributes associated with variables, (2) various operators to be applied to vector and array operands, and (3) a large number of nonstandard operators. Moreover, because APL allows dynamic data handling and because it is an interactive language, data type checking, subscript checking and text editing are to be performed at execution time.

In order to overcome inefficiency in APL software interpreter due to these features, some microprogrammed APL interpreters, similar to IBM Hassitt's machine, are experimentally implemented on a microprogrammed computer since 1975 in Japan. Various quantitative evaluation data about firmware effectiveness in an APL interpreter were accumulated.

In 1975, an early experiment on a firmware APL computer[30] was made by T. Motooka et al. at Tokyo University on an experimental machine, PPS1[43]. An APL source text is translated into an intermediate language on a one for one basis by a lexical analyzer written in a microprogram. An intermediate language is composed of identifiers, operators, constants and brackets. The order of elements for a statement is same in the internal representation. The interpreter is written in microprograms and APLs. Both the lexical analyzer and the interpreter are implemented on a microprogrammed experimental computer, PPS1. The authors concluded that the firmware APL computer is much slower than an APL machine in software on scalar operations, but faster on many vector operations.

H. Miyawaki et al. of Himeji Institute of Technology made a firmware APL interpreter[31,33], based on a quantitative analysis[32] of the interpretation part, which is implemented in software on a general purpose minicomputer, HITAC-10. An APL source statement is translated into an intermediate text which is composed of 32-bit text elements followed by an end element as shown in Fig. 10. It was indicated that, in a firmwarization, appropriate functional modules, frequently used to implement an APL interpreter, are to be selected rather than all of an APL statement. As a result of this experiment, it is shown that a firmware interpreter, made of about 4.8-k words microprograms is 6 times faster than a software version.

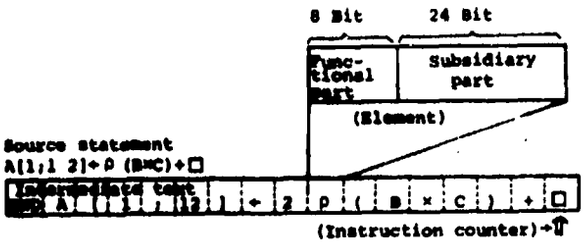


Fig. 10 Source Statement, Intermediate Text and Its Element

Y. Morimoto from Toshiba Ltd. implemented a firmware APL interpreter, APL/EPOS I interpreter[34,35], on an EPOS (Experimental Polyprocessor System) system[44], whose component processor is organized of a universal host microprocessor, PULCE (A high performance universal computing element)[45], dedicated to emulation with powerful microinstruction sets, various kinds of hardware registers and so on. APL source statements are translated into intermediate texts similar to the preceding firmware APL interpreter by a translator written in pseudo APL language (PAPL), which is emulated with microprograms. On the other hand, intermediate texts are interpreted by PAPL and microprograms, and microprograms mainly play scanning for intermediate texts, decision on operation category to be manipulated and execution of basic APL operators. According to evaluation data, APL/EPOS I interpreter is 100 times faster than a software version, on some APL functions. Also, it is faster than the execution of object codes generated by a compiler.

Moreover, another similar research effort[36] has been carried out on a dynamic microprogrammable computer, QA-1[46], by K. Kinoshita et al. of Kyoto University. Various unique experimental results will be obtained because of many special QA-1 features, e.g. hardware stacks, low-level parallel processing capabilities due to using four ALUs and tag minipulation functions.

PASCAL Machine

The use of a structured high-level language, PASCAL, is increasing due to its high portability, programmer/execution-efficiency and compactness of language processing system. At the same time, in order to effectively execute PASCAL programs, PASCAL machines, such as PASCAL Microengine of Western Digital Corp., have appeared.

T. Furuya of ETL experimentally implemented a concurrent Pascal Machine[37] on the multiprocessor system (ACE)[42], based on P.B. Hansen's Concurrent Pascal Machine. An interpreter to execute Concurrent Pascal Machine (CPM) instructions and a Kernel to supervise parallel processes were made with both PDP-11/45 instructions and CPM oriented language (C-language) which were emulated with ACE system microprograms. C-language consists of conventional machine instructions like PDP11/45 and frequently used CPM instructions. In order to parallelly execute multiple processes on a multiprocessor system, process synchronization instructions and I/O operations, having a process schedule function, are introduced to the Kernel with the aid of an ACE synchronization module. As a result of the experiment, various valuable evaluation data were shown, and great decrease in overhead time was attained by parallel execution of processes and efficient process switching.

Other Research Efforts on High-level Language Machine Design Problems

In addition to high-level language machine implementation efforts described earlier, a number of other research efforts related to high-level language machine design problems have been made. The intermediate language architecture of a high-level

language machine is one of major keys for successful implementation. Some evaluations[38,39] on this problem were accomplished. Moreover, the problem of a multilingual high-level language machine was considered[40].

Summary

High-level language machines in Japan were surveyed. Generally speaking, much of them are at the stage of fundamental and experimental research completion. In the future, the appearance of regular commercial high-level language machines and the confirmation of their effectiveness will be desired.

References

PL/I

1. M. Sugimoto PL/I reducer and direct processor, Proc. ACM 1969 PP. 519 - 538

FORTRAN

2. S. Takahashi An experiment of a firmware FORTRAN processor, Proc. IPSJ 14th Programming Symposium 1973 PP. 201 - 208 (in Japanese)
3. O. Miwa, et al. FACOM 230-75 array processor system, Fujitsu Vol. 29 No.1 1978 PP. 93 - 128 (in Japanese)
4. K. Uchida, et al. The FACOM 230-75 array processor system, 3rd USA-JAPAN Computer Conference 1978 PP. 369 - 373
5. Y. Umetani, et al. An analysis on applicability of the vector operations to scientific programs and the determination of an effective instruction repertoire, 3rd USA-JAPAN Computer Conference 1978 PP. 331 - 335
6. R. Takanuki, et al. Some compiling algorithms for an array processor, 3rd USA-JAPAN Computer Conference 1978 PP. 273 - 279

BASIC

7. Y. Nagai, et al. An experimental study on a firmware-implemented high level language machine [I], EC Monograph of IECE of Japan EC74-29 1974 PP. 73 - 84 (in Japanese)
8. M. Yamamoto, et al. An experimental study on a high level language machine with specialized hardware modules [II], EC Monograph of IECE of Japan, EC74-30 1974 PP. 85 - 92 (in Japanese)
9. K. Kumano, et al. A quantitative evaluation of a high level language machine NEC R & D No.50 1978 PP. 30 - 41
10. M. Yamamoto An evaluation of a high level language machine architecture, Tech. Memo of WGARC of IPSJ Vol. 75-9 1975 (in Japanese)
11. T. Maruyama A firmware BASIC interpreter [I], Proc. of 19th Nat'l Conf. of IPSJ 1978 PP. 23 - 24 (in Japanese)
12. T. Maruyama A firmware BASIC interpreter [II], Proc. of 20th Nat'l Conf. of IPSJ 1979 PP. 31 - 32 (in Japanese)

COBOL

13. M. Yamamoto, et al. A COBOL machine architecture, Proc. IPSJ 19th Nat'l Conf. 1976 PP. 307 - 309 (in Japanese)
14. M. Yamamoto, et al. Design of a COBOL oriented high level language machine, Proc. of 3rd USA-JAPAN Computer Conference, 1978 PP. 417 - 421
15. M. Yamamoto, et al. A COBOL machine design and evaluation, Proc. of International Workshop on High-Level Language Computer Architecture, 1980

LISP

16. T. Shimada, et al. LISP machine and its evaluation Tech. Memo of WGARC of IPSJ No. 74-7 1974 (in Japanese)
17. T. Shimada, et al. A LISP machine and its evaluation, J. IECE Vol. J59-D No.6 1976 PP. 406 - 413 (in Japanese)
18. Y. Yamaguchi, et al. A LISP machine on the ACE system, EC Monograph of IECE of Japan Vol. EC76-13 1976 PP. 67 - 75 (in Japanese)
19. T. Yamaguchi, et al. Dynamic measurements of LISP programs on a virtual machine, J. IECE Vol. J61-D No.8 1978 PP. 517 - 524 (in Japanese)
20. M. Nagao, et al. Machine architecture and micro-instruction structure of a LISP machine NK3, EC Monograph of IECE of Japan Vol. EC77-17 1977 PP. 67 - 78 (in Japanese)
21. M. Nagao, et al. LISP machine NK3 and its performance evaluation, Tech. Memo of WGSYM of IPSJ Vol. 7-4, 1979 (in Japanese)
22. T. Taki, et al. Experimental LISP machine, Tech. Memo of WGARC of IPSJ Vol. 32-3 1978 (in Japanese)
23. K. Taki, et al. Experimental LISP machine and its evaluation, Tech. Memo of WGSYM of IPSJ 1979 (in Japanese)
24. T. Usuki, et al. Experimental LISP machine on multi-processor system, Proc. IPSJ 19th Nat'l Conf. 1978 PP. 27 - 28 (in Japanese)
25. T. Usuki, et al. LISP machine implementation on multi-microprocessor system, Tech. Memo of WGARC of IPSJ Vol. 33-4, 1979 (in Japanese)
26. H. Yasui et al. Parallel processing of EVLIS machine, Proc. IPSJ 20th Nat'l Conf. 1979 PP. 183 - 184 (in Japanese)
27. H. Yasui et al. Dynamic behaviour of parallel processing on a LISP program and a system configuration of EVLIS machine, Tech. Memo of WGSYM of IPSJ Vol. 10-4 1979 (in Japanese)
28. M. Ida, et al. Lisp machine based on a microprocessor: ALPS/I, J. IPSJ Vol. 20 No.2 1979 PP. 113 - 121 (in Japanese)
29. E. Goto et al. FLATS, A machine for numerical, symbolic and associative computing, Proc. of the 6th Annual Sumposium on Computer Architecture 1979 PP. 102 - 110

APL

30. T. Motooka, et al. A firmware APL machine, Proc. IPSJ 16th Nat'l Conf. 1975 PP. 109 - 110 (in Japanese)

31. N. Miyawaki, et al. An APL interactive processing system in firmware, Proc. IPSJ 16th Nat'l Conf. 1975 PP. 111 - 112 (in Japanese)
32. N. Miyawaki, et al. Analysis of the interpreter of the APL interactive processing system and the points to implement in firmware, J. IPSJ Vol. 19 No.5 1978 PP. 390 - 397 (in Japanese)
33. N. Miyawaki, et al. Effectiveness of firmware in an APL interpreter, J. IPSJ Vol. 20, No.2 1979 PP. 172 - 178 (in Japanese)
34. Y. Morimoto, et al. Operator processing in an APL interpreter, Proc. IPSJ 17th Nat'l Conf. 1976 PP. 547 - 548 (in Japanese)
35. Y. Morimoto Implementation methods of a firmware APL interpreter and its evaluation, Tech. Memo of WGSYM of IPSJ Vol. 9-2 1979 (in Japanese)
36. K. Kinoshita A firmware APL processor on the [illegible], Proc. IPSJ 20th Nat'l Conf. 1978 PP. 29 - 30 (in Japanese)

PASCAL

37. T. Furuya Concurrent pascal machine on multiprocessor system (ACE), EC Monograph of IECE of Japan, EC 78-30 1978 PP. 1 - 10 (in Japanese)

OTHER DESIGN PROBLEMS

38. M. Arisawa Compilers with intermediate code, and intermediate code machines, EC Monograph of IECE of Japan EC 74-26 1974 PP. 45 - 52 (in Japanese)
39. K. Tanaka, et al. Design and evaluation of intermediate language for firmware intermediate-language machines, Proc. of 16th Nat'l Conf. of IPSJ 1975 PP. 133 - 134 (in Japanese)
40. Y. Kitajima, et al. A microprogrammed implementation of high-level language oriented multi-processor system, EC Monograph of IECE of Japan EC 74-28 1974 PP. 63 - 71 (in Japanese)

GENERAL

41. M. Yamamoto, et al. A microprogrammed computer design and evaluation system, Proc. 1st USA-JAPAN Computer Conference 1972 PP. 139 - 154
42. M. Iizuka et al. ACE - A new modular computer architecutre, Proc. 2nd USA-JAPAN Computer Conference 1975 PP. 36 - 41
43. T. Motooka, et al. Polyprocessor system: PPS-1, Information processing Vol. 15 No.7 1974 PP. 557 - 564 (in Japanese)
44. M. Maekawa, et al. Experimental polyprocessor system (EPOS) - Architecture, Proc. of The 6th Annual Symposium on Computer Architecture 1979 PP. 168 - 195
45. M. Iizuka, et al. Development of a high-performance universal computing element-PULCE, Proc. AFIPS National Computer Conference Vol. 47 1978 PP. 1255 - 1264
46. H. Hagiwara, et al. Hardware organization of a low level parallel processor, Proc. of IFIP Congress 77 1977 PP. 855 - 860

Reflections on a High Level Language Computer System or Parting Thoughts on the SYMBOL Project

David R. Ditzel[†]

Bell Laboratories
Murray Hill, New Jersey

William A. Kwinn[†]

Hewlett-Packard
Ft. Collins, Colorado

ABSTRACT

The SYMBOL system is the prime example of the actual construction and use of a high level language computer. It is unique in the architecture, the instruction set, and the language. This paper attempts to summarize some of the lessons learned from the machine during the last eight years of its use. Comments are made on the high level instruction set, and how the descriptor and tag mechanisms affected the system. Several of the processors are discussed, including the automatic memory management and the hardware implemented operating system. The difficulties encountered in debugging the hardware and the software are compared.

Introduction

One of the most radical computer architectures of the last decade was unveiled in 1971 with the announcement of the SYMBOL[1,2] computer system. The prime goal of the SYMBOL research project was to demonstrate with a full-scale working computer that a procedural general-purpose programming language and a large portion of a time-shared operating system could be implemented directly in hardware, resulting in a marked improvement in computational rates.[3] A further goal was to show that such a task could be mounted by a relatively small group of people in a reasonable amount of time through the use of appropriate design tools and construction techniques. The announcement and initial papers on this computer system were made at a time when it was not yet fully operational, and was being moved to Iowa State University for final debugging, evaluation and use. After arrival at ISU the computer was made fully operational, and was used in a programming environment.

It would be nice if a definitive statement could be made neatly categorizing all of the successes and failures of the project. Unfortunately, such data was remarkably difficult to collect, project members still disagree on many issues. Part of the problem in evaluating SYMBOL was that the machine was radically different from traditional computers in so many ways that a controlled comparison was practically infeasible. Nevertheless, we feel it is important to state our opinions; it should be understood that the following comments are personal observations by the authors, based upon four years of daily contact with the SYMBOL machine. In defense of the original designers of the machine, we feel it necessary to reiterate that SYMBOL was intended as a learning device, rather than as a commercially viable product.

†Work done at Iowa State University under NSF grant GJ33097X

Background

The roots of SYMBOL go back as far as 1964, when it was decided by a group of engineers at Fairchild's research facility in Palo Alto, California that the future of integrated circuit technology dictated the use of hardware for traditional software functions. The design of the system was, and still is, a unique example of a completely top down design. It was felt that existing programming languages had been influenced too heavily by the underlying hardware, and that valuable programmer time was unnecessarily being spent performing functions such as memory management because of unreasonable computer architectures. A high level language computer was seen as an answer to reducing rising software costs.

One of the first tasks tackled was the specification of a new programming language(SPL)[4,5] along the lines of ALGOL 60 and PL/I, but without underlying machine influences. The language was designed for processing character oriented data that could be variable in type, shape and size. Rigid type and size declarations that would normally aid a compiler were omitted from the language as they were seen to burden the user; conversions and space management were handled automatically by SYMBOL's hardware. Structures of arbitrary shape were to be explicitly representable in the language. A top down design was derived from the language specification and the desire to support multiple users in an interactive environment. Part of the research effort was to probe the limits of hardware; even such traditional software functions as the text editor were put in hardware. The system was designed so that a user could walk up to a cold computer, turn it on, and have all the functions necessary to begin programming in a high level language using virtually no system software. The resources needed to design this complex hardware were substantial. A computer aided design system[6,7] was developed to check timing and loading, to do placement and wire routing, and to maintain a system for documenting the circuitry of more than 20,000 packages.

At the time that the fabrication of SYMBOL was completed and debugging began, the semiconductor industry was in a recession and a managerial decision was made not to continue the project through a second design that Iowa State University was to have received for evaluation. Instead ISU obtained the original machine from Fairchild in 1971, through a grant from the National Science Foundation, for the purpose of bringing the machine to full operation so that the unique ideas of the architecture could be more fully documented and evaluated. At ISU the machine was brought into useful operation by 1973. Work on the system software and hardware was done by a group of about six people, mainly graduate students. Funding for the project terminated in 1978, and shortly afterwards hardware failures forced the machine to be permanently decommissioned.

Experience with a High Level Instruction Set

The SYMBOL instruction set[8,9] reflects the SYMBOL Programming Language with almost a one-to-one correspondence between tokens in the source and the object code. The hardwired Translator takes a source program and generates an internal postfix representation to be executed by the Central Processor. All operators are generic; the types of operands are determined from the descriptors and type tags associated with each identifier or constant. The instruction set is aesthetically appealing in its simplicity. There are approximately fifty instructions, only six of which require an address field. All references to identifiers are made with an instruction that contains the address of the identifier's descriptor. Constants may appear in-line and are always tagged. The advantages of the instruction set would appear to be its semantic conciseness and uniform mechanism for referencing data.

Code compaction

There are several problems with the high level nature of the instruction set, only a few of which are specific to SYMBOL. The high level and postfix stack orientation of the instruction set were expected to give good code compaction. Closer examination however revealed that SYMBOL's code was much less compact for typical programs than on traditional machines such as the IBM 360 or PDP-11. Several factors account for this poor code density. A substantial fraction of the object code consisted of non-functional "end of statement" operations, debugging links pointing to the source program and No-Ops. Code density was also lost due the fact that opcodes, which are 1 byte in length, could be placed only in the first or fifth bytes of the eight byte word, thus wasting three bytes for each opcode that did not require an address field. The Translator contributed to the problem by producing extremely poor code, at times even replicating non-functional instructions. The strict one-to-one correspondence between source and object code resulted in the absence of many instructions that could have been useful in optimizing for common special cases. Examples of such instructions would be increment, set to zero, and append a character. The unusual memory structure also hindered code compaction by prohibiting any address calculations, thus precluding space saving using relative addressing techniques. The lesson learned was that code compaction does not necessarily result from high level instructions, and that factors of two or three in code density can be lost without careful integration of the instruction set, compiler technology and the memory structure.

High Level Instructions and Interrupt Handling

An unexpected lesson was that there are times when instructions can be at too high a level. Because of the variable length operands and high level operations, hundreds or even thousands of memory references could be required to execute a single instruction. This had rather severe consequences on interrupt handling (page fault, disk servicing, user interrupt, process switch, etc.). Proper interrupt handling requires the ability to stop execution, handle the interrupt, and then resume execution of the original instruction at the point of the interrupt. For efficiency reasons it is important to be able to stop execution of an instruction (without completion), save all state information active in the processing of the instruction and resume execution at or near the point of interruption rather than to restart execution of the instruction from the beginning. For a high level algorithm, the state information that must be saved can be rather large. A large fraction of SYMBOL's design bugs were the result of the failure to save all the necessary state information. This type of bug was extremely difficult to track down, as the fatal interrupt was often generated non-deterministically from combinations of disk interrupts, clock time-outs or users pressing interrupt buttons. Another problem was the inability to save all the necessary information for particular stages of the algorithm. These oversights were eventually fixed, sometimes at the expense of storing state information at "convenient checkpoints". Restarting at such checkpoints repeated needless work after task shutdowns, and worse, caused hundreds of times more state saves than were necessary; this degraded system performance perhaps as much as 20%.

Optimization

Code optimization in SYMBOL would be difficult to achieve because of the generalized nature of the operations. The addition of lower level instructions could have allowed optimization of many special cases. For example, incrementing a variable on SYMBOL could take over a dozen memory references due to its stack mechanism and indirection through descriptors. The uniform referencing to data structures meant that a compiler could not optimize accessing for special cases; in particular a tremendous performance penalty was paid with SYMBOL because the memory structure made it impossible to perform traditional indexing and address calculations. Even if such indexing were possible, there would be an incompatibility because of the inability to do binary arithmetic for addressing on the decimal only machine.

Descriptors and Tags

Because SYMBOL was one of the few examples of a descriptor based machine and a tagged architecture, a few comments are appropriate. Operand and instruction tagging was useful in catching occasional machine errors where, for a number of reasons, a memory reference returned an incorrect value. There were never any instances where data could possibly be mistaken for program or vice versa; this did in fact report many machine errors that might have gone undetected in a traditional machine. Tags were also of great benefit in debugging and in developing sophisticated software debugging tools.

Descriptors had an even stronger impact on SYMBOL, both positive and negative. Descriptors were invaluable in efficiently implementing the dynamic typing present in the language and in the benefits provided for debugging tools. On the other hand, implementing recursion in the SYMBOL Programming Language was a task left to system software, and turned out to be extremely inefficient. A simple test of Ackermann's function would show SYMBOL to be at least three orders of magnitude slower than traditional machines. The main problem was that the descriptors for the entire procedure had to be copied upon a recursive call if the descriptors themselves might be modified in the call -- a virtual certainty in SYMBOL.

Need for a Systems Language

One of the problems with the SYMBOL language and instruction set was that they were not efficient for lower level tasks common to systems programming. The support tools on SYMBOL could have been more effectively supported though a systems oriented language such as BCPL,[10] BLISS,[11] or C.[12] While inefficiencies in short lived

user programs could be tolerated the same can not be said for system software. The SYMBOL Programming Language turned out to be inappropriate for systems programming. It is recommended that even on computers that intend to support only one user language, a significant effort should go into supporting an underlying systems language. Addition of a few lower-level instructions could have made SYMBOL an effective multi-language system.

System Software and the Hardwired Operating System

The functions of a complete time-shared operating system were implemented directly in hardware by the System Supervisor,[13] aided by the Memory Controller, Memory Reclaimer, Channel Controller, Drum Controller, and Input/Output Processor. System software was intended only to handle certain exceptional conditions, but in fact was used to a much greater extent than the designers originally foresaw. Substantial efforts of the research team were spent on developing loaders, text editors, improved diagnostics, debugging packages, library routines and a file system. This software was seen as essential to make the system/user interface tolerable. System software accounted for several thousand lines of code by the end of the project. Much of the success of this software was due to the foresight of the designers in providing "hooks" in the hardware for software intervention, allowing the system to retain some flexibility despite its hardwired implementation.[14]

Two important questions are answered by SYMBOL concerning the benefits derived from implementing major parts of an operating system in hardware. First, it would seem that the overall design costs of developing a hardware implemented operating system are much higher than an equivalent software implementation; the desire to lessen the cost of developing an operating system was not achieved. *Software* costs were reduced, but *overall* costs were not. Traditional software bug fixes were merely exchanged for a "Request for Hardware Modification" sheet, the bound RFHMs were over four inches thick -- and accounted only for changes after the system was delivered "debugged" to ISU!! The second and more positive point is that the implementation of the hardwired operating system seems to have been very successful from a performance and programming standpoint. Though the inflexibility of the hardware often prohibited changes towards more "modern" operating system concepts, the implementation was very successful in terms of the original design goals. Using hardware for heavily used functions such as process scheduling, virtual memory management, memory allocation, and scheduling of multiple processors seems to have been a wise tradeoff. It was also shown that complex hardware can be successfully interfaced to the software part of the operating system. In terms of the overall design, SYMBOL deserves recognition as a successful Operating System Machine as much as it does for being a High Level Language Machine.

A Tale of Two Processors

While hardwired implementation of high level functions has its merits, a look at two of SYMBOL's processors might prove insightful. Perhaps the most striking aspect of SYMBOL to a user was the amazing speed at which programs were compiled (70,000 to 100,000 statements per minute). The SYMBOL Translator[15] is probably the only example of a compiler implemented entirely with random logic. The Translator is perhaps the most amazing of SYMBOL's processors, not only because of its tremendous speed of compiling but also in that it worked at all. One of the benefits of this tremendous translation speed was that no object files were saved. This was an advantage in saving storage space and in insuring that object programs always reflected the current source program.

We do not wish to imply, however, that such speeds are generally obtainable from a hardwired compiler and a high level instruction set. The performance figures of SYMBOL's Translator are somewhat misleading in that the speed came primarily from two other factors. First, the SPL language[4,5] had a grammar designed to be easy to parse. Non-optimal code was generated in one pass with backpatching and without the need for building compile-time data structures. The high translation speed could not be expected in a proper implementation of a compiler for SPL or more complex programming languages. Second, the Translator did almost nothing more than crude code generation or assembly. Error diagnostics were next to non-existent, though in the majority of cases syntax errors in programs were detected. Our experience suggests that compilers should only be constructed using a high level programming language. Compiler complexity can perhaps be attacked more successfully by using modern compiler writing tools[16,17] than by developing high level instruction sets. The poor design of the Translator was undoubtedly due in large part to the low-level implementation the designer was forced to work with and the infantile state of compiler technology in the early 1960's.

Debugging the Translator hardware was extremely difficult, as register level flow charts and wire lists proved to be a totally inadequate form of documenting the conceptual process of translation. In no way could the design, implementation and debugging of the SYMBOL's Translator have been cost effective compared to a compiler programmed in a high level language. The hardware dedicated to the Translator was not cost effective, as the logic was rarely in use and a similar function could have been performed by the Central Processor. Perhaps a more reasonable tradeoff would have been to provide the Central Processor with special purpose hardware to aid with the various translation functions. This would have had the added benefit of allowing special purpose hardware to be used for other functions in addition to translation.

Even more than the Translator, the I/O Processor suffered from the rigidity of a hardwired implementation. To offload the Central Processor, the I/O Processor contained a hardwired text editor that ran extremely quickly. Unfortunately the pushbutton operated editor was so difficult to use and so primitive that all on-line editing was done in the Central Processor with software text editors. The strict separation of the I/O Processor and the Central Processor did not allow the primitives in the hardwired text editor to be shared by the software text editors.

Two lessons are evident. First, essential utilities of a system such as a text editor and compiler need the ability to change and grow, both to correct bugs and to add new features. The hardwired approach did not allow the possibility for this growth. The functional division was at too gross a level, e.g. the specialized hardware in the Translator provided an all or none service. Second, special purpose hardware is made flexible by modularizing primitive operations so they can be controlled by the software. If the sequencing of the primitives in the I/O

Processor had been controllable by software accessible by the Central Processor, performance of the software editors might have been much closer to that of the hardwired text editor. Much of the problem of SYMBOL was that the designers thought they knew how users would want to use the machine. When this view was changed even slightly, the hardwired nature of the Translator, Editor and operating system locked the user into a mold he did not want to be in.

Memory Management: A Case of Strange Bedfellows

One of SYMBOL's unique features was its complex memory organization. SYMBOL provided direct hardware support both for a paged virtual memory and for dynamic data structures. The SYMBOL hardware supported the allocation, deletion and manipulation of storage strings. These storage strings were constructed by linking together eight-word groups. Linked lists of such storage strings were used to represent tree structures which were accessed in SPL as heterogeneous arrays. The sizes and shapes of these structures were dynamically variable.

The designers of SYMBOL foresaw and attempted to mitigate the adverse interaction of SYMBOL's unique combination of memory management and virtual memory. They realized that particular machine functions had characteristic memory access patterns. For example, the source code was used in program editing but not at all during execution. In program compilation, source code and object code were scanned only once, whereas the name tables were scanned repeatedly. Hence, the designers decided that each page should be used for a single purpose and that page lists would be maintained to segregate the pages according to their use. When memory was allocated, the crude usage class for the needed space was specified by the hardware. This usage class determined which page list the system would consult to find the needed space. SYMBOL maintained three separate page lists: one for source code, another for object code, and the third for all other needs. Once any space on a page was allocated, the page was inserted on the appropriate page list. Henceforth, that page would only be used for further allocations of space of the same usage class. This scheme worked well for program editing and for constructing name tables and object code at compile time. However, at execution time, all data accessing involved one page list, so there was no advantage to this scheme at that time.

It would have been worth while to experiment with adding more page lists to SYMBOL -- lists of pages used solely for the stack, for temporaries, or for large structures. This likely would have limited the scattering of these objects by restricting them to a segregated set of pages. Unfortunately, implementation of additional page lists would have required extensive modifications throughout SYMBOL's Central Processor, and hence was never actually tried.

In SYMBOL, a single large structure could come to occupy small portions of a large number of pages. There was no mechanism for compacting these structures. Modifications to the memory allocation strategy attacked the problem by preventing some of any reclaimed space on each page from being found, except for expansion of structures which already occupied a portion of that page. This was known as the Space Available List(SAL) Threshold technique.[18] Measurements taken on SYMBOL programs which had had significant paging activity indicated that this approach reduced the number of page faults dramatically. The greatest benefits were realized when one sixth to one fourth of each page was reserved for the above-mentioned expansion.

Experiments were performed reducing SYMBOL's page size from the built in 2K-bytes per page as low as 256 bytes per page. The use of smaller pages usually reduced the paging activity for a fixed main memory size. This technique worked whenever severe scattering was encountered, regardless of its origin. Unfortunately, the use of small pages could hurt where sequential access to a large body of code or data was typical. Furthermore, the cost of the overhead associated with a large number of pages could become significant. Although it would have contradicted the declaration-free character of SPL, one cannot help but speculate that the ability to request contiguous allocation of large structures would have reduced paging considerably.

Debugging Software on SYMBOL

An outstanding benefit from the high level nature of the SYMBOL computer was shown in the efficacy of the debugging tools[19,20] produced for the system. Programs were developed to allow the user to examine the state of his program in detail at the source program level. For example, at a user-generated interrupt the programmer could ask the Inquire subsystem where the program was executing and have the statement in execution decompiled for display. The decompilation process was remarkably effective, and generally differed from the original source program only with respect to spaces and redundant parentheses. Since SYMBOL was a descriptor based and tagged architecture, the current types and values of all identifiers in the user's program were known.

There was never any need for a programmer to realize that his program was being translated into an intermediate form for execution. This is one of the strongest points for the claim that SYMBOL was a High Level Language Computer System.[21] In addition to the benefits that the machine offered for debugging, the dynamic type checking mechanisms in the hardware proved very valuable for detecting occasional machine errors such as trying to use instructions as data or vice versa.

Debugging Hardware on SYMBOL

One of the questions the implementation of SYMBOL was supposed to answer was whether or not extremely complex hardware could be designed and debugged. The answer is that complex hardware can be designed and debugged but only through the investment of tremendous effort and time. In 1971 SYMBOL was debugged to the point where it could run simple programs, yet in 1978 bugs were still being found in various processors. The situation appears to be no different from bugs that plague software years after a program is developed, even if it is continuously having bugs removed. The authors' experience with debugging the SYMBOL system and more conventional software projects would suggest that bugs in hardware occur in much the same way that they do in software. However, the problems associated with finding and curing hardware bugs are far more severe.

Changes to hardware are more time consuming than changes to software. Modifications to SYMBOL had to be done with extreme care, changes often had unexpected side effects because the conceptual details of an algorithm were not documented as they might have been

with well commented software. It was not uncommon to cure the symptom rather than cure the problem because of this lack of conceptual documentation. Unlike software, certain changes could not be made because of physical limitations such as the number of bus pins or the number of IC packages that would fit on a board. Hardware errors and bugs were not always deterministic. Because of this non-determinism it was first necessary to ascertain whether a bug was due to an incorrect algorithm or if a circuit was failing because of a bad component.

Any similar scale hardware project must make special efforts to provide the maximum possible effort for developing design and debugging tools. The state of the art in constructing and debugging digital systems is far behind the same technology of software systems. This is probably connected with the limited use of high level engineering systems such as SCALD[22] or DRAW.[23] Computer aided debugging is a necessity. SYMBOL needed the ability to trace and store the last several thousand operations in real time and have the trace information analyzed automatically. The limited trace facility on SYMBOL perturbed the system sufficiently that some errors would go away when traced, and when a problem could be traced reliably it was often beyond the ability of a human to read through hundreds of lines of hex bit patterns to find the offending error.

Von Neumann Realities

SYMBOL is a classic example of a distinctly non-von Neumann architecture. Features that take it out of the von Neumann class are the non-contiguous memory structure, automatic memory management, distinguishability of instructions from data, the self-describing nature of structures, and the high level instruction set. An early paper made the comment that

> as implemented in the SYMBOL hardware, however, any task requiring the variable field length processing and storage or the dynamic structure features of the language should show a considerable performance gain over conventional software/hardware systems. [3]

Experience with SYMBOL suggests that this is probably true, but unfortunately there were not enough tasks of this type.

The reality was that programs on SYMBOL, as on most computers, tended to do relatively simple operations. Arithmetic operations were mainly adding or subtracting very small integers; little use was made of the 99 digit precision controlled arithmetic. Character strings were most frequently only a single character, and rarely exceeded a dozen characters in length. While some use was made of dynamically variable arrays, arrays were almost always homogeneous and remained static once grown. At the machine level, it hurt a great deal that the memory structure and decimal arithmetic processor precluded indexing with address arithmetic. Object code, name tables, and source files were always static objects after their creation; a better storage organization for these would perhaps have been a traditional contiguous linear store. The moral of this story is that the traditional von Neumann computer is perhaps not so ill-suited to the operations actually performed by typical programs. The S?L language and SYMBOL hardware were more powerful than the average user required. Some of SYMBOL's more advanced features could have been implemented by software on a traditional machine to achieve a more cost effective solution to the same problems. Perhaps the conclusions would have been different in another environment, but SYMBOL was not as much an advantage over the von Neumann machine as had been hoped earlier

Microcode

The hardwired nature of the SYMBOL machine is often criticized for its inflexibility. Microcoding has been suggested as an implementation solution that is flexible and still efficient. The understanding of the authors is that during the 60's when technology decisions were being made, ROM's suitable for microcode lacked speed, lacked density, and were prohibitively expensive for the quantities required for SYMBOL. If one were to design the same processors today, microcoding is obviously superior to a random logic implementation. Part of the SYMBOL experiment, however, was to push the limits of a completely hardwired implementation; microcode would not have accomplished this. The significant lessons to be learned from SYMBOL are not whether it should have been microcoded or not, but rather in the lessons learned about system complexity, refinement of complex systems, debugging of complex systems, functional division, and instruction set design. In many instances system software needs to be installation modifiable, a microcode implementation would generally not fall into this category.

Was SYMBOL Really a HLLCS?

It is crucial to note why we consider SYMBOL to be one of the few real High Level Language Computer Systems. The SYMBOL machine, with and only with the software developed for it, meets the HLLCS definition[21] because it:

(1) Uses a high level language for all programming, debugging and other user/system interactions.

(2) Discovers and reports syntax and execution errors in terms of the high level language source program.[20]

(3) Does not have any outward appearance of transformations from the user programming language to any internal languages.

Perhaps the most crucial part of meeting this definition in any system is being able to debug a program at the source language level. The SYMBOL architecture facilitated this with high level instructions that allowed object code to be easily de-compiled back into source, and in the self-describing nature of all data objects that allowed the unambiguous interpretation of any data storage. A High Level Language Computer System is different from and more important than just a machine with a high level instruction set.

Conclusion

The existence of the working SYMBOL computer system clearly demonstrates that a high level instruction set, a compiler, automatic memory management and a major portion of a time shared operating system can be implemented successfully in hardware. Use of the SYMBOL system showed to a lesser degree that the costs of building such a system are not less than building an equivalent system in software; that the ability to evolve a system is perhaps more important than having a very fast functional unit that is never used; that performance gains from hardwired implementation are easily lost.

SYMBOL taught us a great deal about building complex systems. The top down design approach made it necessary for the the entire system to be conceived before any of it was implemented; the results show that this is dangerous. Building complex hardware is prone to the same bugs and fundamental design errors that plague complex software systems. SYMBOL contained many excellent and unique solutions to individual problems but the complex interactions of all of these solutions combined to make the entire system cumbersome and slow. Refinement and iterative improvement are steps that most software systems must go through before reaching acceptable levels of performance and utility; this step was desperately needed with SYMBOL. Performance could have been improved perhaps more than an order of magnitude if many of the known inefficiencies could have been tuned or removed. Despite several negative comments in this paper, the SYMBOL experience was a very positive first step in the design of High Level Language Computer Systems.

References

1. R. Rice and W. R. Smith, "SYMBOL -- A Major Departure from Classic Software Dominated von Neumann Computing Systems," *Proceedings of the AFIPS 1971 Spring Joint Computer Conference*, Montvale, N.J., pp. 575-587, AFIPS Press (1971).
2. W. R. Smith, et al., "SYMBOL -- A Large Experimental System Exploring Major Hardware Replacement of Software," *Proceedings of the AFIPS 1971 Spring Joint Computer Conference*, Montvale, N.J., pp. 601-616, AFIPS Press (1971).
3. G. D. Chesley and W. R. Smith, "The Hardware-Implemented High-Level Language for SYMBOL," *Proceedings of the AFIPS 1971 Spring Joint Computer Conference*, Montvale, N.J., pp. 563-573, AFIPS Press (1971).
4. H. Richards, Jr., "SYMBOL IIR Programming Language Reference Manual," Report ISU-CCL-7301, Cyclone Computer Lab., Iowa State University, Ames, Iowa (1973). NTIS accession number PB-221 378.
5. H. Richards, Jr. and C. Wright, "Introduction to the SYMBOL-2R Programming Language," *Proceedings of the ACM-IEEE Symposium on High-Level-Language Computer Architecture*, New York, Association for Computing Machinery (1973). NTIS accession number PB-228 115/AS.
6. B. E. Cowart, R. Rice, and S. F. Lundstrom, "The Physical Attributes and Testing Aspects of the SYMBOL System," *Proceedings of the AFIPS 1971 Spring Joint Computer Conference*, Montvale, N.J., pp. 589-600, AFIPS Press (1971).
7. M. A. Calhoun, "SYMBOL Hardware Debugging Facilities," *Proceedings of the AFIPS 1972 Spring Joint Computer Conference*, Montvale, N.J., pp. 359-368, AFIPS Press (1972).
8. D. R. Ditzel, "Program Measurements on a High Level Language Computer," *Accepted for publication in Computer* (1979).
9. P. C. Hutchison and K. Ethington, "Program Execution in the SYMBOL 2R Computer," *Proceedings of the ACM-IEEE Symposium on High-Level-Language Computer Architecture*, New York, Association for Computing Machinery (1973). NTIS accession number PB-228 780/AS.
10. M. Richards, "BCPL: A Tool for Compiler Writing and Structured Programming," *Proceedings of the AFIPS 1969 SJCC* (1969).
11. W. A. Wulf, D. B. Russell, and A. N. Habermann, "BLISS: A Language for Systems Programming," *Communications of the ACM* 14(12), pp. 780-790 (December 1971).
12. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey (1978).
13. W. R. Smith, "System Supervisor Algorithms for the SYMBOL Computer," *Digest of Papers, COMPCON72*, New York, pp. 21-26, IEEE (1972).
14. H. Richards, Jr. and A. E. Oldehoeft, "Hardware-Software Interactions in SYMBOL-2R's Operating System," *Proceedings of the Second Annual Symposium on Computer Architecture*, NTIS accession number PB-239 220/AS (1975).
15. T. A. Lalioris, "Implementation Aspects of the SYMBOL Hardware Compiler," *Proceedings of the First Annual Symposium on Computer Architecture*, pp. 111-115 (1973).
16. S. C. Johnson and M. E. Lesk, "UNIX Time-Sharing System: Language Development Tools," *Bell Sys. Tech. J.* 57(6), pp. 2155-2175 (1978).
17. B. W. Leverett, R. D. G. Cattell, S. O. Hobbs, J. M. Newcomer, A. H. Reiner, B. R. Schatz, and W. A. Wulf, "An Overview of the Production Quality Compiler-Compiler Project," Report CMU-CS-79-105, Carnegie-Mellon University (February 1979).
18. W. A. Kwinn, "Memory Management Policies for a Hardware Implemented Computer Operating System," Special Report MCS72-03642-CL7801, Cyclone Computer Laboratory, Iowa State University, Ames, Iowa (1978).
19. D. R. Ditzel, "Interactive Debugging Tools for a Block Structured Programming Language," Report MCS72-03642-CL7802, Cyclone Computer Laboratory, Iowa State University, Ames, Iowa (1978).
20. D. R. Ditzel, "High Level Language Debugging Tools on the SYMBOL Computer System," *1980 Workshop on High-Level Language Computer Architecture*, Fort Lauderdale, Florida (May 1980).
21. D. R. Ditzel and D. A. Patterson, "Retrospective on High-Level Language Computer Architecture," *Proc. of 7th Ann. Symp. on Computer Architecture*, La Baule, France (May 1980).
22. T. M. McWilliams and L. C. Widdoes, Jr., "SCALD: Structured Computer-Aided Logic Design," Technical Report No. 152, Digital Systems Laboratory, Stanford University, Stanford, California (March 1978).
23. A. G. Fraser, "UNIX Time-Sharing System: Circuit Design Aids," *Bell Sys. Tech. J.* 57(6), pp. 2233-2249 (1978).

Appendix. SYMBOL Bibliography

1. O. Agrawal, *Applicability of Buffered Main Memory to SYMBOL 2R Like Computing Structures*, Iowa State University, Ames, Iowa (1974). Ph.D dissertation
2. L. M. Alarilla, Jr., "Storage Linking Techniques for the Automatic Management of Dynamically Variable Arrays," Report ISU-CL-7403, Cyclone Computer Laboratory, Iowa State University, Ames, Iowa (1974).
3. J. W. Anderberg and C. L. Smith, "High-Level Language Translation in SYMBOL 2R," *Proceedings of the ACM-IEEE Symposium on High-Level-Language Computer Architecture*, New York, Association for Computing Machinery (1973). NTIS accession number PB-228 117/AS.
4. J. W. Anderberg "Source Program Analysis and Object String Generation Algorithms and their Implementation in the SYMBOL 2R Translator," Report NSF-OCA-GJ33097-CL7410, Cyclone Computer Laboratory, Iowa State University, Ames, Iowa (1974). NTIS accession number PB-230 614/AS
5. R. W. Black, "Structured Programming in the SYMBOL-2R Programming Language," Special Report ISU-CL-7405, Cyclone Computer Laboratory, Iowa State University, Ames, Iowa (1976).
6. A. C. Bradley, "An Algorithmic Description of the SYMBOL Arithmetic Processor," Report NSF-OCA-GJ33097-CL7301, Cyclone Computer Laboratory, Iowa State University, Ames, Iowa (1973). NTIS accession number PB-222 972.
7. R. F. Bretl, "A Hierarchic Control Structure for User Programs in the SYMBOL System," Special Report ISU-CL-7501, Cyclone Computer Laboratory, Iowa State University, Ames, Iowa (1976).
8. M. A. Calhoun, "SYMBOL Hardware Debugging Facilities ," *Proceedings of the AFIPS 1972 Spring Joint Computer Conference*, Montvale, N.J., pp. 359-368, AFIPS Press (1972).
9. G. D. Chesley and W. R. Smith, "The Hardware-Implemented High-Level Language for SYMBOL," *Proceedings of the AFIPS 1971 Spring Joint Computer Conference*, Montvale, N.J., pp. 563-573, AFIPS Press (1971).
10. Y. Chu, "Significance of the SYMBOL Computer System," *Digest of Papers, COMPCON72*, New York, pp. 33-35 (1972).
11. R. F. Cmelik and D. R. Ditzel, "The High Level Language Instruction Set of the SYMBOL Computer System," *1980 Workshop on High-Level Language Computer Architecture*, Fort Lauderdale, Florida (May 1980).
12. B. E. Cowart, R. Rice, and S. F. Lundstrom, "The Physical Attributes and Testing Aspects of the SYMBOL System," *Proceedings of the AFIPS 1971 Spring Joint Computer Conference*, Montvale, N.J., pp. 589-600, AFIPS Press (1971).
13. M. C. Dakins, "Nonnumeric Processing in the SYMBOL-2R Computer System," Report NSF-OCA-GJ33097-CL7410, Cyclone Computer Laboratory, Iowa State University, Ames, Iowa (1974).
14. D. R. Ditzel, "MASK and FORMAT: Operators for Editing and Formatting," *SIGPLAN Notices* **12**(11), pp. 28-35 (November, 1977).
15. D. R. Ditzel, "Pattern Matching for High Level Languages," *SIGPLAN Notices* **13**(5), pp. 46-55 (May, 1978).
16. D. R. Ditzel, "Interactive Debugging Tools for a Block Structured Programming Language," Report MCS72-03642-CL7802, Cyclone Computer Laboratory, Iowa State University, Ames, Iowa (1978).
17. D. R. Ditzel, "Program Measurements on a High Level Language Computer," *Accepted for publication in Computer* (1979).
18. D. R. Ditzel and W. A. Kwinn, "Reflections on a High Level Language Computer System or Parting Thoughts on the SYMBOL Project," *1980 Workshop on High-Level Language Computer Architecture*, Fort Lauderdale, Florida (May 1980).
19. D. R. Ditzel, "High Level Language Debugging Tools on the SYMBOL Computer System," *1980 Workshop on High-Level Language Computer Architecture*, Fort Lauderdale, Florida (May 1980).
20. H. Falk, "Hard-Soft Tradeoffs," *IEEE Spectrum* **11**(3), pp. 42-43 (Feb, 1974).
21. P. C. Hutchison and K. Ethington, "Program Execution in the SYMBOL 2R Computer," *Proceedings of the ACM-IEEE Symposium on High-Level-Language Computer Architecture*, New York, Association for Computing Machinery (1973). NTIS accession number PB-228 780/AS.
22. P. C. Hutchison, "Extensions to a Block-Structured Programming Language to Support Processing of Symbolic Data and Dynamic Arrays," Special Report ISU-CL-7705, Cyclone Computer Laboratory, Iowa State University, Ames, Iowa (1977).
23. W. E. Jones, "The Role of the Interface Processor in the SYMBOL IIR Computer System," Special Report NSF-OCA-GJ33097-CL7304, Cyclone Computer Laboratory, Iowa State University, Ames, Iowa (1973). NTIS accession number PB-227 454/AS.
24. W. E. Jones, "A Microprocessor-Based Input/Output System for an Interactive Computer," Special Report ISU-CL-7503, Cyclone Computer Laboratory, Iowa State University, Ames, Iowa (1976). Ph.D dissertation

25. W. A. Kwinn, "Memory Management Policies for a Hardware Implemented Computer Operating System," Special Report MCS72-03642-CL7801, Cyclone Computer Laboratory, Iowa State University, Ames, Iowa (1978).

26. T. A. Laliotis, "Implementation Aspects of the SYMBOL Hardware Compiler," *Proceedings of the First Annual Symposium on Computer Architecture*, pp. 111-115 (1973).

27. T. A. Laliotis, "Main Memory Technology," *Computer* 6(9), pp. 21-27, IEEE (September 1973).

28. T. A. Laliotis, "Architecture of the SYMBOL Computer System," in *High-Level Language Computer Architecture*, ed. Y. Chu, Academic Press (1975).

29. G. J. Meyers, pp. 97-147 in *Advances in Computer Architecture*, John Wiley and Sons, Inc. (1978).

30. E. I. Organick, *Proceedings of the AFIPS Workshop on the Influence of Programming Languages on Computer Systems Architecture*, Montvale, N.J., AFIPS Press (1971).

31. R. Rice and W. R. Smith, "SYMBOL -- A Major Departure from Classic Software Dominated von Neumann Computing Systems," *Proceedings of the AFIPS 1971 Spring Joint Computer Conference*, Montvale, N.J., pp. 575-587, AFIPS Press (1971).

32. R. Rice, "The Hardware Implementation of SYMBOL," *Digest of Papers, COMPCON72*, New York, pp. 27-29, IEEE (1972)

33. R. Rice, "A Project Overview," *Digest of Papers, COMPCON72*, New York, pp. 17-20, IEEE (1972).

34. H. Richards, Jr. and R. J. Zingg, "The Logical Structure of the Memory Resource in the SYMBOL-2R Computer," *Proceedings of the ACM-IEEE Symposium on High-Level-Language Computer Architecture*, New York, Association for Computing Machinery (1973). NTIS accession number PB-228 118/AS.

35. H. Richards, Jr. and C. Wright, "Introduction to the SYMBOL-2R Programming Language," *Proceedings of the ACM-IEEE Symposium on High-Level-Language Computer Architecture*, New York, Association for Computing Machinery (1973). NTIS accession number PB-228 115/AS.

36. H. Richards, Jr., "SYMBOL IIR Programming Language Reference Manual," Report ISU-CCL-7301, Cyclone Computer Lab., Iowa State University, Ames, Iowa (1973). NTIS accession number PB-221 378.

37. H. Richards, Jr. and A. E. Oldehoeft, "Hardware-Software Interactions in SYMBOL-2R's Operating System," *Proceedings of the Second Annual Symposium on Computer Architecture*, NTIS accession number PB-239 220/AS (1975).

38. H. Richards, Jr., "Controlled Information Sharing in the SYMBOL-2R Computer System," Special Report ISU-CL-7601, Cyclone Computer Laboratory, Iowa State University, Ames, Iowa (1976). Ph.D dissertation

39. C. L. Smith, C. T. Wright, and R. J. Zingg , "Problems in the Push-Down Stack Approach to the Implementation of High Level Languages," *Digest of Papers, COMPCON76*, New York, pp. 96-98, IEEE (1976).

40. W. R. Smith, "System Supervisor Algorithms for the SYMBOL Computer," *Digest of Papers, COMPCON72*, New York, pp. 21-26, IEEE (1972).

41. R. E. Wolf, "SYMBOL 2-R Compatible Tree Manipulation," Special Report ISU-CL-7602, Cyclone Computer Laboratory, Iowa State University, Ames, Iowa (1976).

42. R. J. Zingg and H. Richards, Jr., "Operational Experience With SYMBOL," *Digest of Papers, COMPCON72*, New York, pp. 31-35 (1972).

43. R. J. Zingg and H. Richards, Jr., "SYMBOL: A System Tailored to the Structure of Data," *Proceedings of the National Electronics Conference*, Oak Brook, Illinois 27, pp. 306-311, National Electronics Conference, Inc. (1972). NTIS accession number PB-221 286.

44. W. R. Smith, et al., "SYMBOL -- A Large Experimental System Exploring Major Hardware Replacement of Software," *Proceedings of the AFIPS 1971 Spring Joint Computer Conference*, Montvale, N.J., pp. 601-616, AFIPS Press (1971).

A CASE AGAINST HIGH-LEVEL LANGUAGE COMPUTER ARCHITECTURE

Harvey G. Cragon

Texas Instruments Incorporated
Dallas, Texas 75265

ABSTRACT

This paper considers the principle motivations for a high-level language architecture, Programmer Productivity, Compiler Simplification, and Run-Time Efficiency. Individually and collectively, these motivations do not represent compelling justification for a departure from conventional architectures. It is suggested that a more beneficial architectural departure is to be found in a lower-level micro architecture instead of a higher-level architecture.

INTRODUCTION

The question of the desirability of a high-level language architecture was asked at the birth of the stored program digital computer by Burks, Goldstine, and von Neumann[1].

> "In general, the inner economy of the arithmetic unit is determined by a compromise between the desire for speed of operation -- a non-elementary operation will generally take a long time to perform since it is constituted of a series of orders given by the Control -- and the desire for simplicity or cheapness of the machine."

Over the years, architectural trade-offs have been made in favor of selective incorporation of complex functions in those architectures where performance was a dominant consideration. Floating point as an elementary operation was provided as a hardware operation in the mid-1950s[2]. A variation of the FORTRAN DO loop was included in the CDC STAR and TI ASC architectures in the 1970s[3]. With vector instructions included as elementary operations, the generation of addresses is overlapped with the operation itself yielding improved performance and a reduction in required memory bandwidth is achieved by the reduction in the number of instruction fetches.

A view has been introduced into the discussion of elementary operation selection. This view is an observation that a "semantic gap"[4] exists between the programming language and the language which the computer actually executes. The existence of a gap is an invitation to close the gap.

A recurring idea is the high-level language architecture which directly executes a selected language. SYMBOL[5,6] is this type of architecture as is the recently discussed Ada processor by Intel[7]. For many reasons, these architectures, labeled "Type C" by Myers[8], are deemed inefficient. Most proposals today for a high-level architecture embrace some intermediate language[9] as the language to be accepted by the computer.

Proposals for high-level language architecture are based on achieving three improvements:

1. Programmer Productivity
2. Compiler Simplification
3. Run-Time Efficiency

PROGRAMMER PRODUCTIVITY

Unfortunately, the observation has been made that closing the gap will have a significant positive impact on programming cost. This has had the result of drawing attention away from the real problem of selecting elementary operations. I believe that this argument proceeds as follows:

1. The best performance and the minimum code space results when a problem is programmed in assembly language.
2. Poor performance and code space result if a high-level language is used.
3. Programmer efficiency is improved if a high-level language is used.

Thus, a carefully selected intermediate execution language, which can be compiler generated, will give good performance, reduced code space, and increase programmer productivity.

Programming costs are a function of the language and the quality of the support functions provided. It should make no difference in programmer productivity whether the support functions are provided in hardware or software.

Assistance in program debug is a benefit cited for a high-level language architecture[10] which should reduce programming cost. I believe that there is a lesson to be learned today from the support systems provided for microprocessors. Program development is moving into a cross support mode. More and more programs are developed on a host which is not the computer on which the

program will execute[11]. One reason for this is that powerful debug tools can be provided in the development software. Only a very small subset of these tools could be provided in the hardware of a high-level language architecture, software support would still be needed. Relating execution errors during development to the source program is enhanced more with software tools then with a menger set of hardware capabilities.

COMPILER SIMPLIFICATION

A benefit frequently advanced for a high-level architecture is that a well-selected set of intermediate level language significantly reduces the complexity of the compiler. This is hard to understand. It can be argued that these compound elementary operations of the intermediate language can be defined as macro subroutines which the compiler can easily produce. These macros can then be interpreted by the machine. Again, this becomes a question of cost and performance. This "soft" intermediate level language architecture yields all of the desirable compiler characteristics as does a "hard" architecture. The Burroughs B1700[12] is an illustration of this point. Cohen and Francis[13] describe another system which executes on conventional microprocessors.

I will not argue that the specification and use of an intermediate level language is not beneficial for compiler creation. I do argue that this language, in total, should not be implemented in hardware. For those cases where an intermediate language seems beneficial to the compilation process, interpretation of this language is completely feasible, although slow in execution. The benefits of reduced code space, including the interpreter, generally are realized.

RUN-TIME EFFICIENCY

I perceive that the semantic gap has become highly visible because of two factors. First, the non-computational overhead of structured programming is increasing the run time of our programs, and second, the execution of operating system functions is also consuming a highly visible amount of CPU time. In both of these cases, the root problem stems from the lack of a few elementary operations selected to support these functions, not a closing of a semantic gap.

Myers[14] provides an interesting comparison of the concepts of PL/I and the support provided by the S360. I believe that in every case cited by Myers, the issue resolved itself into the need for the compiler to generate a body of code which implements the PL/I concept. This is an issue of elementary operation selection and the cost performance of the computer.

The cost performance of a computer having more complex elementary operations is of real concern. Let me examine the reduction in memory bandwidth resulting from the inclusion of vector instructions. Myers[15] describes the case of two 100 by 100 element fixed binary arrays which are to be added together. A programmed loop would require 40,004 memory references for instructions and 30,003 for data, a total of 70,007. A single vector instruction would require only 30,001 (30,000 for operands and one instruction). An alternative to this is found in computers such as the CDC 7600, which has a program buffer cache. This architecture requires only eight references to main memory for the instructions and 30,000 references for the data. Vector instructions are not needed to reduce memory bandwidth if instruction buffering and high execution rate is provided for the elementary operations.

The use of compound elementary operations can reduce the storage requirements for instructions due to the instructions' higher information content. In Myer's example, the number of instruction bytes is reduced from 274 to 13. This is an impressive reduction! However, if the program represents 20% of the total memory requirement, for example, the compound elementary operations can yield, at best, a 20% reduction in required total memory space. This small memory savings may not be worth the increased cost of the CPU.

Compound elementary operations to enhance run-time cost effectiveness are provided at a cost in hardware, logic, and microcode. The justification of this cost depends upon the number of times the function is executed in a program; frequent use justifies, occasional use does not. Figure 1 illustrates this point. The higher the cost of providing a hardware macro, the larger the use factor must be to achieve a breakeven cost.

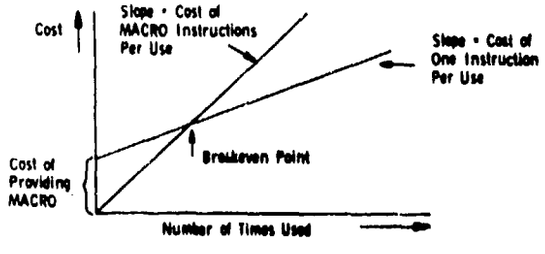


Figure 1

Computer architects can quickly select most of the elementary operations of their design. The inclusion of more complex or compound elementary operations requires knowledge of the intended use of the computer. Care must be exercised that static and dynamic statistics collected on programs run on a unique computer reflect the true nature of the problem and exclude the characteristics of the computer[16]. For example, code used for run-time checks will not be identified with the higher purpose of the code. Nevertheless, choices are made and computers are

designed and built, which are improvements over prior designs.

For a computer which must be multilingual, that is, can be programmed in many languages, great care must be exercised in the selection of compound elementary operations which will be useful for all the languages. The result of implementing the intermediate language in hardware can be a loss of generality. An intermediate language for COBOL is not likely to be the same language for FORTRAN or PASCAL. And what does one do when Ada becomes popular? Will the intermediate language support the new programming language efficiently?

Figure 2 illustrates the problem which is created as the language implemented by the hardware approaches the programming language, closing the semantic gap. In a conventional processor, the high-level language is compiled into machine language which is interpreted by the hardware. As the machine language approaches the programming HLL, the machine languages will diverge and become two or more different machine languages if the semantic gaps are completely closed.

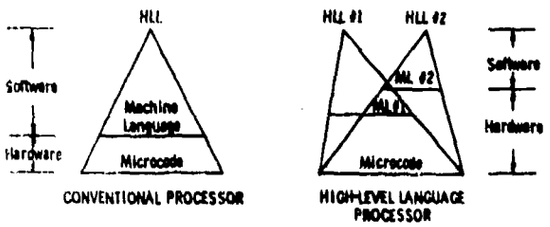


FIgure 2

Kavipurapu and Cragon[17] are conducting a search for common elements and their frequency of use in FORTRAN, COBOL, and PASCAL to see if there are a few compound operations which will benefit all three languages. I believe that there is a good chance that a small number will be found that, if implemented in hardware, will substantially improve a computer's code space and execution time. Success in finding a few is not a mandate to implement everything in an intermediate language.

A high-level or intermediate language implemented in hardware is too restrictive and costly. However, selective implementation of a small set of compound elementary operations can substantially improve the performance of a computer. The question facing computer architects today is not high-level language architectures, but architectures which permit the inclusion of selected compound elementary operations which match the use environment at any given time.

A writable control store with program access to sequences of microcode is one technique. This will, in effect, provide for the interpretation of the compound elementary operations by microcode. Substantial improvement in program execution time can result[18,19,20]. The compiler should be able to make a selection of those compound elementary operations which are interpreted by the machine's elementary operations and those which are to be interpreted by microcode. Production runs of a program can further adapt the mix to achieve the fastest execution rate.

A second technique, and one which is attractive for implementation in VLSI, is the use of compound function attached processors[21]. A floating point chip and an FFT butterfly chip which can be attached to a microprocessor are examples. A Decimal String Chip would be useful for a microprocessor executing a heavy COBOL load.

I will concede that there may be a place in computer architectures for the inclusion of hardware employed to improve the reliability of software in execution. The run-time environment creates problems which cannot be anticipated by the compiler or require high checking overhead. This issue should be addressed as a stand-alone issue and should not be combined with the issue of a high-level language architecture.

The ultimate architecture approach was suggested, I believe, by McKeeman in 1967[22].

> "The obvious attack for programmers and hardware people together is to devise language that reflects what we want to do and how we do it (for instance, in parallel) and machine structures effective in handling that language. Let us call this method 'language directed computer design.'"

In the future, the language referred to by McKeeman must mean nonprocedural programming techniques[23,24]. The machine structures will be microprogrammed in nature. The architecture will be capable of either interpreting a "soft" intermediate language or executing a compiled microprogram. With memory becoming the least costly component, compiled microcode will become more and more cost effective. If a lower performance is satisfactory, then the interpreted soft intermediate language can reduce memory cost. I believe that there is no "ideal DEL," there may be a DEL for every nonprocedural language and this DEL can be interpreted on a soft architecture if memory cost is to be minimized.

CONCLUSIONS

A case has not been made for the creation of new architectures which implement high-level or intermediate level languages. All of the benefits can be achieved without the loss of generality by selective implementation of some compound elementary operations in callable micro-

code or attached processors. The ultimate architecture will be a lower-level one, not, as many advocate, a higher-level one.

ACKNOWLEDGEMENTS

The author would like to thank the reviewers for constructive criticism which helped clarify the position taken in this paper. Special thanks go to George Ligler without whose encouragement this paper would not have been completed.

REFERENCES

1. Arthur W. Burks, Herman H. Goldstine, and John von Neumann, "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument," The Institute for Advanced Study, June 1946, p. 1.

2. Saul Rosen, "Electronic Computers: A Historical Survey," Computing Surveys, Vol. 1, No. 1, March 1969, p. 14.

3. C. V. Ramamoorthy and H. F. Li, "Pipeline Architecture," Computing Surveys, Vol. 9, No. 1, March 1977, pp. 85-95.

4. U. O. Gagliardi, "Report of Workshop 4--Software-Related Advances in Computer Hardware," Proceedings of a Symposium on the High Cost of Software, Menlo Park, Calif.: Stanford Research Institute, 1973, pp. 99-120.

5. Rex Rice and William R. Smith, "SYMBUL--A Major Departure from Classic Software Dominated von Neumann Computing Systems," AFIPS SJCC, Vol. 38, 1971, pp. 575-587.

6. William R. Smith, Rex Rice, Gilman D. Chesley, Theodore A. Lalliotis, Stephen F. Lundstrom, Myron A. Calhoun, Lawrence D. Gerould, and Thomas G. Cook, "SYMBOL--A Large Experimental System Exploring Major Hardware Replacement of Software," AFIPS SJCC, Vol. 38, 1971, pp. 601-616.

7. "Intel Takes Aim at the '80s," Electronics, Vol. 53, No. 5, February 28, 1980, pp. 89-95.

8. Glenford J. Myers, Advances in Computer Architecture, John Wiley & Sons, New York, 1978, p. 29.

9. L. W. Hoevel, "Ideal Directly Executed Language: An Analytical Argument for Emulation," IEEE Transactions on Computers, Vol. C-23, No. 8, August 1974, pp. 11-13.

10. Andrew S. Tanenbaum, Structured Computer Organization, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1976, p. 384.

11. Stefano Crespi-Reghizzi, "A Survey of Microprocessor Languages," Computer, Vol. 13, No. 1, January 1980, p. 65.

12. W. T. Wilner, "Design of the Burroughs B1700," AFIPS FJCC, Vol. 41, 1972, pp. 489-497.

13. Harvey A. Cohen and Rhys S. Francis, "Macro-Assemblers and Macro-Based Languages in Microprocessor Software Development," Computer, Vol. 12, No. 2, February 1979, pp. 53-61.

14. Glenford J. Myers, Advances in Computer Architecture, John Wiley & Sons, New York, 1978, pp. 11-13.

15. Ibid., p. 16.

16. G. Jack Lipovski and Keith L. Doty, "Developments and Directions in Computer Architecture," Computer, Vol. 11, No. 8, August 1978, p. 57.

17. Krishna Kavipurapu and Harvey G. Cragon, International Workshop on High-Level Language Computer Architecture, Fort Lauderdale, Florida, 1980.

18. A. M. Abd-Alla and David C. Karlgaard, "Heuristic Synthesis of Microprogrammed Computer Architecture," IEEE Transactions on Computers, Vol. C-27, No. 9, September 1978, pp. 816-827.

20. Tomlinson G. Rauscher and Ashok K. Agrawala, "Dynamic Problem-Oriented Redefinition of Computer Architecture via Microprogramming," IEEE Transactions on Computers, Vol. C-27, No. 11, November 1978, pp. 1006-1014.

21. G. Estrin and C. R. Viswanathan, "Organization of a 'Fixed-Plus-Variable' Structure Computer for Computation of Eigenvalues and Eigenvectors of Real Symmetric Matrices," Journal of ACM, Vol. 9, No. 1, 1962, p. 41.

22. W. M. McKeeman, "Language Directed Computer Design," AFIPS FJCC, Vol. 31, 1967, pp. 413-417.

23. Michael Hammer, W. Gerry Howe, Vincent J. Kruskal, and Irving Wladawski, "A Very High Level Programming Language for Data Processing Applications," Communications of the ACM, Vol. 20, No. 11, November 1977, p. 832.

24. William A. Wulf, "Trends in the Design and Implementation of Programming Languages," Computer, Vol. 13, No. 1, January 1980, pp. 14-22.

DESIGN ISSUES OF HIGH LEVEL LANGUAGE DATABASE COMPUTERS*

David K. Hsiao**

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

ABSTRACT

In this paper, the design goals of direct execution database computers are stated. Using an existing database management software system, the paper attempts to show the replacement of the software system with a hardware database computer may not obtain uniform performance gains and storage savings. This discovery may render the original design goals overly ambitious.

On the other hand, the complicating factors which hinder the gains and savings may contribute to the antique modes of database management of conventional software systems. To this end, the paper attempts to isolate these factors and identify the modes of operation for consideration.

1. DESIGN GOALS

Normally, the effective use of a database system by a user requires the user to be familiarised with the languages of the database computer system. There are essentially two such languages: the database definition language (DDL) and the database manipulation language (DML). DDL allows the user (especially, the database administrator or database owner) to define the logical and physical properties of the database. Logical properties of a database are characterized by the database models used. For example, in the relational model[1], the logical properties of the database consists of attributes and domains (of a tuple), tuples (of a relation), primary keys (to the tuples) and relations (of the database). In the hierarchical model[2], the logical properties consists of field names and values (of a segment), sequence fields, primary and secondary indices (of segments), segments (of a type), types (of a parent-child relationship) and relationships (of the database). Likewise, there are logical properties of CODASYL databases[3]. By defining information entities in terms of logical properties of a database model, the user can capture the information content in the database and make (symbolic) references to the information entities.

DDL also allows the user (especially, the database designer) to define the physical properties of the database. Physical properties of a database are those which deal with units of storage (say, number of pages and page size), kinds of storage (e.g., moving-head disks vs fixed-head disks), storage formats of the logical entities (directory format for indices, pointers for related tuples or segments and encodings for repeated attributes or field names) and access modes (e.g., access by direct address calculation, via intermediate records or by way of directories).

Because modern databases are meant to be shared, the database system must provide concurrent access and multi-user operations. DDL of a modern database system must therefore provide a means to allow the database owner (or administrator) to authorize and validate certain users of his database, define different portions of the database for different users (e. g., by creating different views of the same database), specify the types of control operations permitted or denied on the authorized portions, and place procedures (e.g., programs written by the administrator or owner) at the points of access paths to his database (say, at each file opening time).

On the other hand, the database manipulation language (DML) is primarily concerned with the specification of search, retrieval, update, and processing requirements of the database. Because the use of data models enables the information content to be captured in the database, the modern DML enables the user to address the database by content for search, retrieval, update and processing operations. Content-addressing is accomplished in DML as expressions of predicates. For example, the following is a simple expression of three precicates, namely, a conjunction of an equality predicate, an inequality predicate and a greater-than predicate.

(Type=EMPLOYEE) $\wedge$ (Emp-Dept = TOY) $\wedge$ (Salary > 20,000) which specifies those records of the

*The work reported herein is supported by the Office of Naval Research under contract N00014-75-C-0573 and by Defense Advanced Research Project Agency under contract N00014-75-C-0661.

**On leave from The Ohio State University.

employees who are not in the toy department and have salaries greater than 20,000. By referring to specific attributes, providing the necessary predicates, and specifying the intended operations in DML, the user can manipulate the database effectively at various granularities of the database (i.e., at field or attribute-value pair level, tuple or segment level, relation or segment type level, and relationship level).

The goals of high-level language database machine designers are therefore to be able to come up with high-performance and great-capacity computer architectures which allow direct execution of DDL and DML statements of the user application programs. Direct execution of user programs enables the performance and capacity gains of the new machine to be contributed to the user in terms of high-volume management and quick response which are difficult to achieve in conventional software-laden computers for very large database applications. This difficulty is due largely to the fact that conventional computers are not designed specially for database management. Consequently, very elaborate software for database management must be supported on the computers. The execution of very complex and sizeable database management software tends to deplete system resources and provides inadequate responses to user applications.

Can we design direct execution database computers? In other words, are there complications in reaching our design goals?

2. ISSUES COMPLICATING DIRECT EXECUTION

There are at least two issues which have complicated the design goals of direct execution database computers. One issue is related to DDL; the other is concerned with DML. These two issues may render the direction execution of DDL and DML statements for conventional database management application ineffective.

The most illustrative way to study these complications is perhaps by focusing our attention at a specific database model and a certain high-level language database computer design. Here, we focus on the hierarchical model[2]. We choose the DDL and DML of IBM's Information Management System (IMS) for study[4-7]. Presently, IMS is a widely used hierarchical database management software system. For database computer hardware designs, we choose the database computer (DBC) which has been proposed[8,9] to support, among other database models, the hierarchical database model of databases. However, much of the findings produced in the following sections are valid for other models and machines which although not elaborated here, can be found in[10,11,12,13].

2.1 Execution of DDL Statements for Creating New Databases

Directly executable DDL statements for hierarchical databases must be available so that given the logical properties of a hierarchical database, the DDL statements, upon execution, can automatically generate the physical structure of the database for storage. Furthermore, the physical structure generated must take full advantage of the strong points and new capabilities of the database computer.

Let us review briefly the logical properties of an IMS database and present a (hardware) transformation algorithm (as designed for DBC) which converts the logical organization of an IMS database into a physical structure for database computer storage. We will also mention briefly some strong points and new capabilities of the database computer.

2.1.1 Logical Properties of a Hierarchical Database. An IMS database consists of a number of hierarchically related segment occurrences (or simply, segments), each of which belongs to a segment type. In the example Figure 1, segment type A, the root segment type, has three occurrences. All others are dependent segment types, each having a unique parent segment type and zero or more child segment types. Some relationships among the various segments in our examples are:

A1 is the parent of B1 and G1.
H1, H2 and I1 are children of G1.
J1 and J2 are twins.
H1, H2, I1, J1 and J2 are descendants or dependents of G1.
A1, G1 and I1 are ancestors of J1.

Successive levels are numbered such that a root segment is at level 1. All segment occurrences are made of one or more fields.

An IMS database is traversed in the order: parent to child, front to back among twins and left to right among children. The traversal sequence for the database of Figure 1 is (A1, B1, C1, D1, D2, D3, E1, F1, E2, F2, F3, G1, H1, H2, I1, J1, J2, A2, A3). Notice that the traversal sequence defines a next segment with respect to a given segment. A hierarchical path is a sequence of segments, one per level, starting at the root, e.g., (A1, G1, I1, J2).

2.1.2 Automatic Generation of Storage Structure. An IMS database with the above logical properties can be defined in DDL statements which upon execution transform the database into proper storage format of the database computer (i.e., DBC). Because DBC does not address physical records by locations, location-dependent pointers are not used by DBC for the purpose of facilitating hierarchically related records. Instead, physical records are content-addressed by DBC provided that the content of a physical record is presented as one or more variable length attribute-value pairs, known as keywords. Thus, an IMS database is transformed by considering every IMS segment as a physical record (or, simply, record) composed of keywords.

An IMS segment includes a sequence field whenever it is necessary to indicate the order among the twin segments. Since each segment becomes a record and no address-dependent pointers are allowed, the database computer assigns a symbolic identifier to each segment, identifying it uniquely from all other segments in the database.

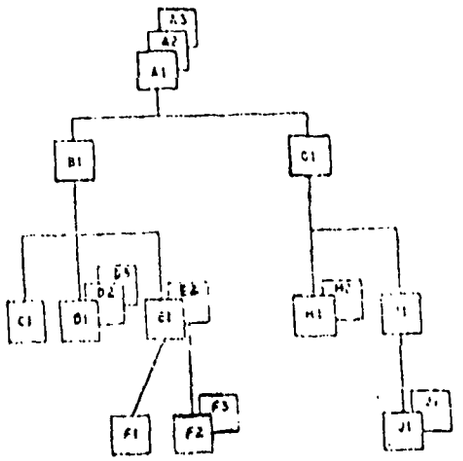


Figure 1. Logical Organization of an IMS Database

The symbolic identifier of a segment S is a group of fields consisting of:

(1) the symbolic identifier of the parent of S, and
(2) the sequence field of S.

Since the sequence fields of different segment types may use the same field name, we may qualify the field name with the segment type.

The creation of a record from an IMS segment can now be accomplished by forming keywords as follows:

(1) For each field in the segment, form a keyword using the field name as the attribute and field value as the value.
(2) Form a keyword of the form <TYPE, segtype> where TYPE is a literal and segtype is the segment type in consideration.
(3) For each sequence field in the symbolic identifier of the segment, form a keyword using the field name (qualified by the segment type) as the attribute and the field value as the value.

For example, for an IMS database shown in Figure 2, the attribute templates of the five collections of records corresponding to the five segment types are shown in Figure 3. Qualified field names such as Prereq. Course # are used to distinguish the same field names, i.e., Course #, among different segment types.

2.1.3 Execution Gain vs Storage Penalty. Due to the simplicity of the transformation algorithm, it is not surprising that DDL statements, which allow the user to specify logically a hierarchical database and provide automatic generation of the database, can be readily realized in the hardware and be executed directly to yield a collection of physical records of keywords for storage. Keywords enable the database computer (DBC) to content-address all records in databases that contain the keywords. Thus, the hardware realization of DML statements indeed utilizes the strong points and new capability of the database computer.

However, if we compare the layouts of Figure 1 and 4, we note that the use of symbolic identifiers to capture the parent-child relationships may increase the storage requirement of the physical records. Furthermore, as the levels of a hierarchy develop the storage requirement of the physical records may increase "exponentially". This is evident by the following observation that for each corresponding dependent segment the physical record must include additional storage space for:

(1) qualifications of the field names, and
(2) sequence fields of its ancestors.

For example, in Figure 3 a physical student record at level three must accomodate the qualified name, Student. Course #. In addition, the student record also must include the sequence field (i.e., the date field) of its parent (i.e., a certain offering record) as a keyword. Since the parent is a child of a certain course record whose sequence field is course number (i.e., Course #), there is a keyword in the student record whose attribute is Course #. The inclusion of course

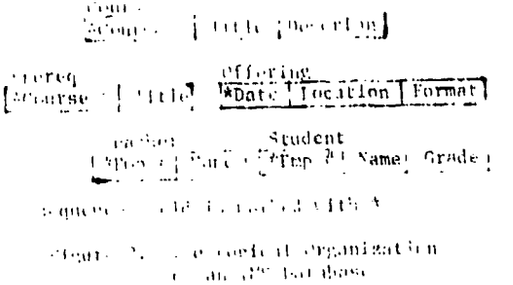
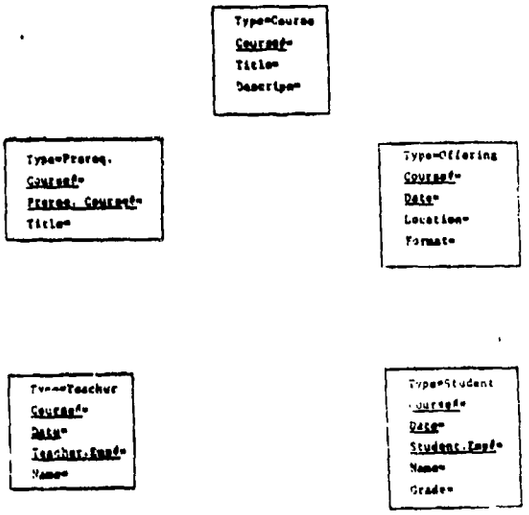


Figure 3. The Logical Organization of an IMS Database

numbers, dates, and qualifications in the student records increase the storage requirement of the hierarchical database considerably. On the other hand, the inclusion of sequence fields as keywords in records eliminates the need of pointer spaces which were necessary in the IMS segments for the purpose of linking all the twins of a given parent sequentially. Despite such trade-off of spaces, analysis has shown[14] that the increase may be 3% per level starting at level 4. Similar findings on storage loss due to new database machine requirement are obtained in relational as well as CODASYL modelled databases.



symbolic identifier is underlined

Figure 3. The Templates of Physical Records of the Segments of Figure 2.

It is not clear whether it is possible to devise a hardware transformation algorithm which is as simple as the one mentioned above and which can yield storage gains. Until such an algorithm is found, direct execution of DDL statements for database creation in the new database computer environment may actually cause a loss in storage.

2.2 Direction Execution of DML Statements for Database Transformation

In IMS, the database manipulation language (DML) statements known as DL/1 calls have the following format

Operation list

where the Operation is one of insert (ISRT), delete (DLET), replace (REPL) and get (GET) calls, and where the list is a number of segment search predicates, at most one per level, which are used to select a hierarchical path. Each segment search predicate is preceded with the name of the segment type. Let us denote the segment search predicate at level i as Si.

After each retrieval or insertion operation, a segment is "established" in the traversal sequence of the IMS database. For a retrieval operation, this segment refers to the segment just retrieved; for an insertion operation, this segment refers to the segment just inserted. Such a segment in the traversal sequence is termed the current position in the database. There are several forms of the get call, each of which returns a single segment. A get-unique (GU) call retrieves a specific segment at level n by starting at the root segment type, finding the first segment at each level i satisfying S_i, and finally retrieving the segment satisfying S_n. A get-next (GN) call starts the search at the current position in the database and proceeds along the traversal sequence satisfying S_i for all i and retrieving the segment satisfying S_n.

We shall illustrate the manner in which get-unique (GU) and get-next (GN) calls are executed by the database computer. Referring back to the IMS database of Figure 2, let us suppose that the DL/1 call to be processed is:

```
GU  Course    (Title = 'MATH')
    Offering  (Location = 'CAMBRIDGE')
    Student   (Grade = 'A')
```

This asks for the first Student segment of the database which satisfies the predicate Grade ='A', and which has a parent segment Offering with Location = 'Cambridge' whose parent, in turn, is a Course segment with Title = 'MATH'. The call is executed as follows:

(1) Starting with the first segment search predicate i.e., Title = 'MATH', the Course segments which satisfy the predicate are retrieved by utilizing the query formulated by the machine
((Type = COURSE) ∧ (Title = MATH))
and are sorted by the machine according to the value of their sequence field, i.e., by the attribute Course #.

(2) If no Course segment exists, then the DL/1 call is unsuccessful. Otherwise, the first Course segment is found and designated as the current Course segment.

(3) The Offering segments are then retrieved with the predicate Location= 'CAMBRIDGE' and sorted by their sequence field, i.e., date. If the sequence field of the current Course segment is (Course #, C), then the query used by the machine for this content-addressing is
((Type = OFFERING) ∧ (Course # = C) ∧ (Location = CAMBRIDGE)).

(4) If no Offering segment exists, then the current Course segment is removed and control is transfered to Step 2. Otherwise, the first Offering segment is designated as the current Offering segment.

(5) The Student segments are then retrieved with predicate Grade = 'A' and sorted by their sequence field, i.e., by Emp #. If the sequence field of the current Course segment is (Course #, C) and that of the current Offering segment is (Date, D), then the query used by the machine for this round of content-addressing is
((Type = STUDENT) ∧ (Course # = C) ∧ (Date = D) ∧ (Grade = A)).

(6) If no Student segment exists, then the current Offering segment is removed and control is transferred to Step 4. Otherwise, the first Student segment is designated as the current Student segment.
(7) The DL/1 call is successfully executed and the current Student segment is returned.

It should be noted that at this point that the content of the work space of the machine established by the above GU call may be used to execute the next DL/1 call, for example, to retrieve the next student who has an A grade in a math course offered in Cambridge. This is depicted by the following get-next (GN) call:

```
GN  Course    (Title = 'MATH')
    Offering  (Location = 'CAMBRIDGE')
    Student   (Grade = 'A')
```

In this case, the relevant segment may already be present in the work space of the machine. The current Student segment is removed and control is transferred to Step 6 given for the GU call.

On the other hand, if the GN call is:

```
GN  Course    (Title = 'MATH')
    Offering  (Location = 'CAMBRIDGE')
    Student   (Grade = 'F')
```

then only existing Course and Offering segments may be used. However, it is necessary that the next Student segment returned should not precede the current Student segment in the traversal sequence. Hence, if the sequence field of the current Student segment is (Emp #,E), that of the current Offering segment is (Date, D), and that of the current Course segment is (Course #, C), then the following machine query is used for content-addressing the next set of Student segments:

((Type=STUDENT) ∧ (Course #=C) ∧ (Date=D) ∧ (Emp # ≥ E) ∧(Grade=F))

The previously existing Student segments are removed and control is transferred to Step 6 given for the GU call.

Finally, if the GN call is

```
GN  Course    (Title = 'HISTORY')
    Offering
    Student
```

then no currently existing segments are useful. Hence, new sets of segments must be retrieved, one set for each level.

2.2.1 Performance Gains Resulted from Direct Execution. From the above discussion, it is not surprising to learn that directly executable database manipulation (DML) statements of the following types of transactions will produce the "best" performance for the database computer over the conventional software-laden IMS system.

Transaction Requirement:

(1) Find all segments satisfying given predicates.
(2) The predicate at the root level does not involve the sequence field.
(3) No predicate is given at any intermediate level.

Example: Find all those students who failed a mathematics course regardless of the location at which the course was offered.

```
    GU   Course    (Title = 'MATH')
         Offering
         Student   (Grade = 'F')

Loop GN  Course    (Title = 'MATH')
         Offering
         Student   (Grade = 'F')
         GO TO Loop
```

Let N be the number of root segments (i.e., courses). All of the root segments satisfying the predicate are content-addressed. For each of these root segments, all y of its third level twins satisfying the predicate are then content-addressed. We also assume that these third level segments (i.e., those students who received grade F) are scattered evenly. The relative performance is charted in Figure 4. The entries of the chart are computed as the ratio of page accesses (to IMS segments in the old software-laden environment) to block accesses (to physical records in the new database computer environment).

Due to very large content-addressable block size (approximately 1/2 megabytes) and relatively small sequential-addressable page size (about 2 kbytes), this type of transaction may yield one or two orders of magnitude of performance gain over the conventional system.

2.2.2 Where are the Performance Gains? Now let us consider another type of transaction as follows:

Transaction requirement --
(1) Find a single segment satisfying the given predicates.
(2) A predicate involving the sequence field is given at root level.

Example: Find the student with employee number 50, taking a CIS 211 course in Columbus. We note that course numbers are sequenced.

```
GU  Course (Course # ='CIS 211');
    Offering (Location =
              'Columbus')
    Student (Emp # = 50)
```

The performance gains of this type of transaction are charted in Figure 5. It is disappointing to note that the performance of the database computer for this type of transaction is not much better than the conventional software-laden system.

Note: The analytical data in the table are extracted from [14].

N = the number of root segments.

y = the number of twins (i.e., children segments).

n = the number of segments satisfying a given predicate.

	N = 100			N = 1000			N = 10000		
n / y	1	16	64	1	16	64	1	16	64
10	36	29	21	216	78	35	720	397	152
20	52	44	41	232	96	55	1,128	507	132
40	44	42	40	146	69	47	938	322	115
80	43	41	40	96	55	44	591	190	78
160	47	46	45	79	54	48	388	133	68

Table Entry = No. of Accesses to Parts of DMS / No. of Accesses to Content-Addressable Blocks of DBC

Figure 4 Performance Gains Measured in Terms of Accesses to Database Storages

Note: The analytical data for the following tables are extracted from [14].

N = the number of root segments.

y = the number of twins.

y / N	100	1000	10000
10	2.33	2.33	0.58
20	4.00	4.00	1.71
40	4.40	4.40	3.44
80	4.67	4.67	4.20
160	5.47	5.47	5.17

Figure 5. Performance Gains of Certain Types of Transactions

2.2.3 Performance Gains vs. Transaction Types. By comparing the examples presented in the previous two sections, it is evident that the new hardware of the database computer will not yield significantly better performance over the software system, if the user transaction demand records in a sequential manner and receive them one record at a time. On the other hand, if for a user transaction, the demand is of high volume and the search criteria of the demand are made of predicates which require content-addressing instead of sequential accessing, then the strong points of the database computer hardware can indeed yield high performance. Ideally one would want to come up with a design of high-performance and great-capacity database computer which can provide effective and efficient solutions to either low-volume and sequential database manipulation or the high-volume and content-addressable database manipulation. Such a design is not in sight.

3. CONCLUDING REMARKS

Direct execution of existing high-level database definition and manipulation language constructs may not be desirable. The undesirability is due to the lack of good database computer design for uniform gains in storage requirement and transaction execution. In other words, special-purpose database computers may not be able to bring about the high hope of anticipated throughput gains which has been the design goal of the database computers in the first place.

Nevertheless, database computers which are capable of directly executing database definition and manipulation language constructs will stay. Their impact will be twofold. First, database application programming will change. The change will primarily be prompted by the advanced features provided by the machines which are not otherwise adequately available in conventional software systems. For example, security and integrity checks and concurrency controls can be made more effectively and efficiently introduced as hardware mechanisms. The use of high-volume and content-addressable search and update for very large databases is another need for hardware realization These advanced features will allow existing databases to migrate to a new database machine environment with newly written application programs. On the other hand, there is not much that the new machine can improve for the old application programs. However, with some interfacing software, the existing application programs can still be run on the new environment without the need of program conversion. It is hoped that in the long run the database application will be dominated by the newly written application programs.

Secondly, the presence of the database machines will have an important impact on the future development of database definition and manipulation languages. Despite their claim of data independence (i.e., devoid of database software and hardware implementation issues), the languages were designed with certain known processing modes and underlying technology of the time. As a new technology with a high degree of parallelism and content-addressability, the database computer will require new database definition and manipulation languages to be highly concurrent and associative. Furthermore, the new languages should have an integrated approach to the specification and control of security and integrity checks of database access and update. Thus, the study of database computer design will also prompt our investigation of new DDL and DML for the computers.

ACKNOWLEDGEMENT

The excellent research environment provided by Professor Michael Hammer on the Project on Very Large Databases and in the Laboratory for Computer Science at Massachusetts Institute of Technology is greatly appreciated. The author was also benefitted by several discussions with Professor Hammer regarding the paper and by Jai Menon for his careful reading of the paper.

REFERENCES

[1] Chamberlin, D.D. "Relational Database Management Systems," ACM Computing Surveys, 8, 1, (March 1976) pp. 43-66.

[2] Tsichritzis, D. C. and Lochovsky, F. H. "Hierarchical Database Management, "ACM Computing Surveys, 8, 1, (March 1976) pp. 105-124.

[3] Taylor, R. W. and Frank, R. L., "CODASYL Database Management Systems," ACM Computing Surveys, 8, 1, (March 1976) pp. 67-104.

[4] IBM, Information Management System/Virtual Storage (IMS/VS) Version 1, General Information Manual, GH20-1260-4.

[5] IBM, Information Management System/Virtual Storage (IMS/VS) Version 1, Application Programming Reference Manual, SH20-9026-4.

[6] IBM, Information Mangement System/Virtual Storage (IMS/VS) Version 1, System Programming Reference Manual, SH20-9027-4.

[7] IBM, Information Mangement System/Virtual Storage (IMS/VS) Version 1, System/Application Design Guide, GH20-9025-4.

[8] Banerjee, J., Baum, R. I., and Hsiao, D. K. "Concepts and Capabilities of a Database Computer," ACM Transactions on Database Systems, 3, 4, (December 1978), pp. 347-384.

[9] Banerjee, J., Hsiao, D. K., and Kannan, K., "DBC -- A Database Computer for Very Large Databases", IEEE Transactions on Computers, C-28, 6, (June 1979), pp. 414-429.

[10] Banerjee, J. and Hsiao, D. K. "The Use of a Database Machine for Supporting Relational Databases, "Proceedings of 4-th Workshop on Computer Architecture for Non-numeric Processing (Syracuse, 1978) Available from ACM.

[11] Banerjee, J. and Hsiao, D. K. "Performance Evaluation of a Database Computer in Supporting Relational Databases, "Proceedings of 4-th International Conference on Very Large Databases, (West Berlin, 1978), Available from IEEE Computer Society.

[12] Banerjee, J. and Hsiao, D. K. "A Methodology for Supporting Existing CODASYL Databases with New Database Machines," Proceedings of ACM '78 Conference, (Washington, D. C. 1978), available form ACM.

[13] Banerjee, J., Hsiao, D. K., and Ng, F. K. "Data Network -- A Computer Network of General-Purpose Front-End Computers and Special-Purpose Back-End Database Machines," Proceedings of the International Symposium on Computer Network Protocols, (February 1978) pp. D6-1 - D6-12. Available from the University of Leige, Belgium.

[14] Banerjee, J., Hsiao, D. K., and Ng, F. K., "Database Transformation, Query Translation, and Performance Analysis of a New Database Computer in Supporting Hierarchical Database Management," IEEE Transactions on Software Engineering, SE-6, 1, (January 1980).

HASHING HARDWARE AID ITS APPLICATION TO SYMBOL MANIPULATION

Tetsuo Ida

Information Science Laboratory
Institute of Physical and Chemical Research
2-1, Hirosawa, Wako-shi, Saitama 351, Japan

Abstract

An architecture of implemented hashing hardware to be used in symbol manipulation is presented. The major components of the hashing hardware are a hash addressing unit and hash table memories which can also be used as main memory of the system. The hardware makes use of parallel read-out and comparison mechanisms of logic-in memory banks. Basic hashing algorithms such as search, insertion and deletion of keys are realized by microprogram control. Performance improvements of ranging 9 - 13 times are obtained over pure software hashing. The application techniques of hashing hardware to symbol table manipulation, property list handling and set operations are given. The advantage of hashing over associative memories in these applications are also discussed.

1. Introduction

Hashing plays an important role in speeding up table look-up operations. It is extensively used, not only in the traditional language translation, i.e. assembling and compiling, but in symbol manipulation at large, e.g. formula manipulation[1], execution of a Lisp dialect[2], and associative processing[3].

Although hashing is the fastest among known methods in the table searching of N items in terms of computational complexity ($O(1)$ compared with $O(\log N)$ of binary search, for example), a constant time factor due to calculation of hash address sequences is not small in software hashing and in some cases, hashing gives way to alternative techniques. Moreover, to avoid rapid degradation of the performance, the table utilization must be limited to far less than that of the total capacity, say 70-80 %.

To overcome these difficulties, we proposed parallel hashing schemes in which n independent hash address sequences are used to access a hash table organized as a b by P two-dimensional array (b columns, to be called memory banks, are accessed in parallel)($n \leq b$) (cf. Fig. 1), and presented performance analyses.[4,5] The results of the analyses assured us of the average execution time of less than 1.18 successful table look-ups with $n=b=4$, or even 1.05 with $n=b=32$ until the load factor of the table gets as high as 0.9.[4]

Based on the analyses, we realized a parallel hashing scheme on an experimental system, to be used for symbol manipulation. In sections 2-5, we discuss the architecture and the performance of the implemented system.

The fact that basic hash table look-up operations can be done with speed comparable to single indirect addressing encourages more extensive use of hashing in new areas of applications. In section 6, we explain how several important algorithms in symbol manipulation are speeded up by the hashing hardware.

2. Initial Design Considerations

Our problem domain is symbol manipulation where tables (data bases) to be searched are taken in main memory and accessed by hashing algorithms such as given in chapter 4 of Knuth.[6]
Our approach is

(1) to build into memory-CPU interface parallel mechanisms of (hash) addressing and data (key) comparison,
(2) to incorporate hardware logic to compute hash addresses into the address formation unit in CPU,

and

(3) to replace the hashing control sequencing (traditionally done by software) by faster logic, i.e. microprogramming.

Several variations of hashing algorithms are known with regards to key collision and deletion handling, apart from the choice of hash functions. We summarized below our considerations on the two issues. For detailed discussion, see papers.[5,7]

Research supported in part by grants in aid from Ministry of Education (No. 479039) and Kurata Research Foundation

Open addressing vs. chaining methods for collision resolution

- When bits required for chaining are rightly taken into account, overall performances of the two are nearly equal.
- The open method is more amenable to parallelism of memory accesses than chaining.

Hence, the open addressing method is selected for our implementation.

With or without key deletion

- Traditional application of hashing such as symbol table manipulation in language translation may not require handling of key deletion, since a symbol table is discarded as a whole when compilation (or assembling) is over.
- However, in the advanced application to be discussed in section 6, key deletion handling is indispensable.
- Among the key deletion algorithms based on the open addressing method, an efficient method developed in [7] requires extra hardware resource in memory (collision number counters in each memory word).
- In our implementation, it is expensive to incorporate extra bits in each word without losing the compatibility with the target computer architecture.

The above considerations lead us to adopt a key deletion algorithm which makes use of three states of a memory word, i.e. 'deleted' (all 1), 'empty' (all 0) and 'occupied' (bit patterns other than the above two bit patterns).

The dificulty with this algorithm is that the 'deleted' words accumulate after repetitions of key deletions and insertions. It causes degradation of the performance, especially unsuccessful searches. We need a clean-up operation of the hash table; i.e. to reclaim 'deleted' words that are no longer in collisions with other keys and to turn them into 'empty' state, relocating keys, if necessary. Without collision number counters, this operation must be performed with the aid of software (rehashing all the keys in the table) in conjunction with garbage collection. The hardware must have a function for monitoring the performance in order to determine when to initiate the garbage collection, however.

3. Description of the Hashing Hardware

Figure 2 shows our experimental system incorporating the hashing hardware unit (HU). It is the implementation of the model in Fig. 1 with n=1 and b=4 in the case of single-length (16 bit) keys. The hashing hardware consists of two parts; hash addressing unit (HAU) and hash table memories (HM). The conventional ALU (16 bits) is microprogram controlled. Without HAU, the system can emulate an existing mini-computer (particularly suited for PDP 11). With the hashing hardware, the instruction repertoire of the processor is augmented with the hashing instructions given in Table 1.

HAU is further divided into three parts; hash address generator (HAG), hash code generator (HCG) and hash table descriptor unit (HTDU), as shown in Fig. 3. HCG is used to generate, out of a key k bit patterns (hash code) which are then input to HAG for the generation of a hash address sequence $\{h_i\}$. HAG implements the following generation algorithm (cf. Fig. 3):

Let c and c' be the hash code, and P be the size of a hash table (cf. Fig. 1). P should be a prime number. To generate h_0 and Δh, we use a mask value 2^{m-1} which satisfies the relation $2^m - 1 > P > 2^{m-1}$.

$h_0 \leftarrow c \wedge (2^m-1)$, $\Delta h \leftarrow c' \wedge (2^m-1)$
if $h_0 \geq P$, $h_0 \leftarrow h_0 - P$
if $\Delta h \geq P$, $\Delta h \leftarrow \Delta h - P$
if $\Delta h = 0$, $\Delta h \leftarrow 1$
for i=1, 2, ..., P-1
$h_i \leftarrow h_{i-1} + \Delta h$
if $h_i \geq P$, $h_i \leftarrow h_i - P$

HM's are realized by logic-in-memory cards, each having 32 k bytes of memory. They are interfaced to common bus (Unibus) (hence accessed as main memory via memory management unit (MMU)), and have following functions;

- parallel read operations of HM1-HM4 which are invoked by HAU,
- pattern matching capabilities, which detect 'deleted', 'empty' states, and key matches.

Hash table descriptor unit (HTDU) in Fig. 3 contains 256 table descriptors and each provides hash table base, size, and the other auxilliary information to be used in HAG, microprogram control unit and ALU. The descriptor of each hash table can also be used to generate an 18 bit address without the use of MMU.

The hashing control is realized by microprogram and its algorithm is discussed in the next section.

4. Basic Hashing Algorithms

Given key k, let k_i's be the simultaneously read-out key from bank i, for i=1,2,...,b. We define following signals to be used in the microprogram control unit;

$$M = m_1 \vee m_2 \vee \ldots \vee m_b$$

$$E = \bar{M} \wedge (e_1 \vee e_2 \vee \ldots \vee e_b)$$

$$D = \bar{M} \wedge (d_1 \vee d_2 \vee \ldots \vee d_b)$$

where

e_i is the result of the comparison of k_i and 'empty',
d_i is the result of the comparison of k_i and 'deleted',
and
m_i is the result of the comparison of k_i and k.
e_i, d_i and m_i are generated in memory bank HMi.

We should note that the comparisons are performed in parallel and that the results (M, E and D) are available immediately after the completion of the key read operations.

Algorithm S (key search)

Instruction HSR is implemented by this algorithm.
Step 1. Set $i \leftarrow 0$
Step 2. Compute a hash address h_i.
Step 3. Access the hash table.
(M, E and D are available at the end of this step.)
Step 4. If M then return the matched position.
If E then terminate the algorithm.
(key k does not exist in the table.)
Otherwise, set $i \leftarrow i+1$, and goto Step 2.

The key deletion algorithm is similar to Algorithm S; replace the first line of step 4 above with "If M then put 'deleted' in the matched position". Instruction HSD is used to execute the deletion algorithm.

The key insertion algorithm which corresponds to HSI is as follows:

Algorithm I (key search and insertion)

Step 1. Set $i \leftarrow 0$.
Step 2. Compute a hash address h_i.
Step 3. Access the hash table.
Step 4. If M then the algorithm terminates.
(Key k already exists.)
If $E \wedge D$ then put k in the 'deleted' position,
and terminate the algorithm.
If E then put k in the 'empty' position
and terminate the algorithm.
If D then set $t \leftarrow$ the 'deleted' position,
set $i \leftarrow i+1$, and goto step 5.
Otherwise, set $i \leftarrow i+1$, and goto step 2.
Step 5. Compute a hash address h_i.
Step 6. Access the hash table.
Step 7. If M then terminate the algorithm.
(Key k already exists.)
If E then put k in position t
and terminate the algorithm.
Set $i \leftarrow i+1$
Go to step 5.

Instruction HNI is used to insert a new key that is known to be non-existent in the hash table. Therefore, the algorithm for HNI is only to repeat the table look-ups until either E or D becomes true.

Execution of the hashing instructions is interrupted when the number of table look-ups exceeds the pre-specified value (steps not shown in the above algorithms). Counting the number of interrupts, the hashing software can monitor the performance of the table look-up operations of a particular hash table; thus we can tell when to invoke the clean-up operation as discussed in section 2. Returning from the interrupt and restarting the instruction is performed by instruction HRTI. Instructions on 'virtual' keys are discussed in section 6.

Key types

The hardware has to cope with multiple-length keys, since the keys are often strings of characters, complex data structures, etc. The operation of HU is not affected by the atrribute of the bit pattern (data type) other than the length. The basic lengths are 'single' (16 bits), 'double', and 'quadruple'. Longer keys are treated either as 'virtual' keys (cf. section 6) or as lists. Hash tables are created to be one of the above types, 'pair' (i.e. pair of a single length key and the associated value) or 'virtual'. The type information is put in the descriptor (obtained from the descriptor) by instruction PTHT (GTHT). This type information is used to invoke appropriate micro code at the exution time of HSR, HGV etc.. Note that for 'double' keys, the hash table appears as two-bank (b=2), and for 'quadruple' keys, as one-bank (b=1).

5. Evaluation of the Performance

Figure 4 is the timing chart of HSR operating on 'single' key. The actual clock periods for t_0, t_1, and t_2 in Fig. 4 are approximately 300, 400 and 1000 ns respectively, and therfore the estimated execution time (excluding the fetch and decode time) of HSR in the case of successful search is $1.6+1.3i$ micro sec, where i is the number of hash table accesses. i depends upon the load factor of the table and the number of memory banks. The values of i based on the theoretical analysis are given in references.[4,5] In the parallel hashing schemes, i is equal to 1 mostly, unless the hash table is heavily loaded.

Talbe 2 shows the timing of typical runs which make use of HSR. We can observe the performance enhancement by a factor of ten over the software hashing. Similar improvements of the performance are observed in the case of the other hash instructions.

6. Application of the Hashing Hardware

Although the hashing hardware is designed to be general as far as possible, in this paper we only give following applications. This is because these are used in existing software systems and the effectiveness of use of hashing is already established. The hardware replacement of the hashing software algorithm will greatly speed up the operations as observed in section 5.

(1) symbol table manipulation in assemblers and compilers,
(2) property list handling,[8]
(3) creation of a unique copy of data structures to enable fast equality checking,[2,9]
(4) as a special case of (3), hash 'cons' in Lisp for the sharing of sub-data structures and fast equality checking,[2]
(5) set operations.[9]

Symbol table manipulation

Figure 5 illustrates data structures of the symbol tables to be used in conjunction with HU. In Fig. 5, HT1 is the 'pair' type hash table. When the key is 16 bit, the key itself is put in the key part of the hash table. Longer keys are accomodated as a pointer to some appropriate entry of another hash table (e.g. when a key is 'double', a pointer to an entry of HT2 is placed in HT1.)

Property list handling

A property list is a Lisp terminology.[10] An implementation method as given in reference[10] relies on sequential search of lists. The method discussed here is a speed-up version of property list handling using hashing. For example, the Lisp code (GET OBJECT ATTRIBUTE) may be executed (interpreted) as

```
HSR     t1,a      ; a points to a double-word key
                  ; consisting of pointers to
                  ; atoms OBJECT and ATTRIBUTE,
                  ; and t1 denotes a hash table
                  ; number.
                  ; This instruction searches for
                  ; a Lisp cell constructed by
                  ; hashed cons(OBJECT, ATTRIBUTE)
BNE     UNSUC     ; If not in the hash table,
                  ; unsuccessful search
                  ; (result in r)
MOV     r,a
HGV     t2,a      ; t2 is the 'pair' type
                  ; hash table, where the value
                  ; associated with
                  ; (OBJECT ATTRIBUTE) is stored.
UNSUC:
```

Creation of unique copy of complex structures

In general, complex structures cannot be treated directly by HU, unless it is built up of uniform structures such as lists in Lisp. It should be formatted so that HU can handle it. One way to handle the complex structure is to make an abbreviated key (p.543 in Knuth[6]) or v(virtual)-key[11] out of it. How to make the v-key is in the realm of software. To treat a v-key as a proper hash key is that of hardware. In treating a v-key, we should note that:

- creation of a v-key out of a complex structure is many-to-one mapping,
- hence, HU has to cope with the situation of multiple key matches.

The search algorithm in a v-key differs from Algorithm S in the following points:

1. When a v-key match occurs, it saves the current hash status $(a_i\ d_i\ m_i\ h_i\ \Delta h)$, and returns the pointer to r-key*1 (performed by instruction HGR).
2. The associated software checks whether r-keys match.
3. If r-key match occurs, the search ends successfully.
   Otherwise, the search restarts from the next point where it is suspended after restoring the hash status (performed by instruction HGRN)
4. When E*2 becomes true, the search terminates unsuccessfully.

As a special case, we consider the case that the key itself is again a pointer to a hash table. This is the case where a set is implemented. Figure 6 shows the data structure. The search algorithm is as follows:

1. Compute the v-key using a symmetric hash function, g
   i.e. g(x,y)=g(y,x), since the order of elements of a set is insignificant.
2. Use HGR and find the v-key match,
3. If E then terminate the algorithm (unsuccessful search).
4. Use HSR to test the matches of each element of the hash tables.
5. If all the elements match, terminate the algorithm, otherwise find the v-key match by HGRN and goto 3.

---

*1 When necessary, we use term 'r(real)-key' to denote the key other than v-keys to clarity the difference.

*2 Strictly speaking, E is not the same as that defined in section 4, since the scan of signals $(a_i m_i d_i)$ may start from the bank different from 1, and since multiple match may occur.

We should note that since HU is used recursively, we need to save the contents of the temporary storage in HU (i.e. Δh, h_i), besides status e_i d_i and m_i in the v-key processing (execution of HGR and HGRH). Hence, we have duplicate of registers in HU actually; ones for r-key hashing and the others for v-key hashing.

In the case of lists, we can do without v-keys. HT3 in Fig. 5 illustrates the shared linked list constructed by unique 'cons' by hashing.

7. Architectural Comparison with Alternative Techniques

To summarize the applications discussed, we see that hashing is used essentially in three ways; (1) associative retrieval and (2) construction of a unique copy of a data structure for fast equality checking, and (3) as a consequence of (2), sharing the sub-data structures in constructing complex structures.

Associative retrieval by hashing is based upon the sigle-hit property of keys. This operation could be performed by associative memories such as surveyed by Yau and Fung.[15] However, we consider hashing more advantageous in our problem domain for the following reasons:

- Hashing is based upon conventional RAMs (Random Access Memory chips), which are simpler in structures at gate level by at least one order of magnitude than associative memory chips e.g. Intel 3104, not to mention the cost performance.
- Furthermore with the same level of semiconductor technology RAMs are faster than associative memories; hence in many applications hashing is faster than associative processing based on associative memories.
- Larger scale implementation is possible with our hashing scheme; the size of the table is limited only by address space of the main memory.
- Full capability of host CPU can be utilized in conjunction with hash table manipulation with no additional hardware cost, since hash tables are realized in main memory.
- Hence the capability of associative retrieval is easily incorporated into existing architecture as shown in previous sections.
- Variety of data structures can be used in hashing, since they are realized in RAMs (main memory) whereas in associative memories data structures would be subjected to hardware memory word configuration.

As for the second and third usage of hashing, corresponding efficient algorithms (e.g. set operations) based on associative memories would be difficult to develop. Different approaches to these applications would be necessary.

8. Concluding Remarks

We have shown how hashing can be implemented by hardware and given some illustrative examples of its use.

The architecture shown in Fig. 2 reflects the basic requirements for the hashing hardware as given in [7]. It also reflects the design compromise imposed by practical considerations for the experimental system; such as cost-performance, compatibility with the existing system, dimensions of the system, etc.. We briefly discuss the alternatives we could have taken if some of the above limitations were removed.

Let us take the execution of HGR operating on a 'single' key (without key deletion), for example. The average execution time is divided into 41%, 12% and 47% when the load factor is 0.9, for memory accesses, key alignment, and other micro operation, respectively. This ratio indicates that the hashing operations are memory-limited. If faster memories or caches are available, the speed of hashing will be further improved.

The generation algorithm of hash address sequences given in HAG suffers from non-uniformity of h_0 and Δh, when the size of the table, P is not close to the power of 2. The trade-off between speed and the uniformity of the distribution of an initial hash addresses is discussed elsewhere.[12]

Examining the figures in table 2, we can conclude that the choice of n=1 (one HAG) and b=4 (four memory banks) seems to be adequate (see reference,[7] for further discussions). However, with additional hardware, we would have chosen parameters b=8 (sligh increase of the performance will result), or b=4 with each memory card equipped with 64 KB. Then all the memories could be usable for hashing.

Software which makes extensive use of the hashing hardware is not yet completed. Full evaluation of the hardware has to await for the software development. The experience with the design and construction of the hashing hardware will be used to buil a larger system for symbolic algebra.[13] We hope that the instruction repertoire will provide data to standardized the hashing operations both in hardware and software. We also hope that in high level language machines hashing hardware will be incorporated as an integrated unit since hashing is believed to speed up essential search operations in interpreter-based systems such as Lisp and a direct execution machine for high level languages.[14]

Acknowledgement

We thank Prof. E. Goto of University of Tokyo for discussions on application techniques of the hashing hardware to various symbol manipulation algorithms. We also thank Mr. K. Hiraki of University of Tokyo and many people of Mitsui Engineering and Shipbuilding for the help in the implementation of the whole system.

References

[1] Goto, E. and Kanada, Y. Hashing lemmas on time complexity with application to formula manipulation, Proc. ACM-SYMSAC, 1976

[2] Goto, E. Monocopy and associative algorithms in an extended Lisp, Tech. Rept. 74-30, Department of Information Science, University of Tokyo, (1974)

[3] Feldman, J. A. and Rovner, P. D. An Algol-based associative language, CACM, Vol.12, No.8 (1968)

[4] Goto, E., Ida, T. and Gunji, T. Parallel hashing algorithms, Information Processing Letters, Vol.6, No.1 (1977)

[5] Ida, T. and Goto, E. Analysis of parallel hashing algorithms with key deletion, Journal of Information Processing, Vol.1, No.1 (1978)

[6] Knuth, D. E. The art of computer programming, Vol.3, Addison-Wesley (1973)

[7] Ida, T. and Goto, E. Performance of a parallel hash hardware with key deletion, Proc. IFIP Congress (1977)

[8] Kanada, Y. Implementation of HLISP and algebraic manipulation language REDUCE-2, Tech. Rept. 75-01, Information Science Laboratories, University of Tokyo (1975)

[9] Sassa, M. and Goto, E. A hashing method for fast set operations, Information Processing Letters, Vol.5, No.2 (1976)

[10] McCarthy, J. et. al. Lisp 1.5 programmers mannual, MIT Press (1965)

[11] Ida, T. and Goto, E. Parallel hash algorithms for virtual key index tables, Journal of Information Processing, Vol.1, No.3 (1978)

[12] Ida, T. A computer architecture with hash addressing capabilities (in preparation)

[13] Goto, E. et. al. FLATS, a machine for numerical, symbolic and associative processing, Proc. 6th annual symposium on computer architecture (1979)

[14] Chu, Y. Direct-execution computer architecture Proc. IFIP congress (1977)

[15] Yau, S. S. and Fung, H. S. Associative processor architecture - A survey, Computing Surveys, Vol.9, No.1 (1977)

Instruction	Function
HSR	Search key
HGV	Get value of 'pair'
HPV	Put value in 'pair'
HNI	New key insert
HSI	Search and insert
HSD	Search and delete
HGR	Get real-key
HGRN	Get real-key next
HPR	Put real-key
HDX	Delete existing virtual-key
HRTI	Return from hash interrupt
PTHT	Put in hash table descriptor
GTHT	Get from hash Table descriptor

Table 1 List of Hashing Instructions

	case 1H	case 2H	case 1S	case 2S
HSR for 'single'keys	6.1	6.6	5.5x10	8.3x10
HSR for 'double'keys	1.1x10	1.2x10	$1.2x10^2$	$1.7x10^2$
HSR for 'quadruple' keys	1.8x10	2.0x10	$2.0x10^2$	$2.3x10^2$

(in micro sec)

Note:
1. Values are average execution timings when accessing all the keys that are
   1H: filled upto 50% of the table that is initially 'empty'
   2H: filled upto 80% of the table that is initially 'empty'.
Cases 1S and 2S are those obtained by executing equivalent pure software (using standard PDP11 instructions) hashing algorithms on the same machine.
2. Timings include fetch and decode time and interrupt handling time if interrupt occurs.

Table 2 Average Execution Timings of HSR

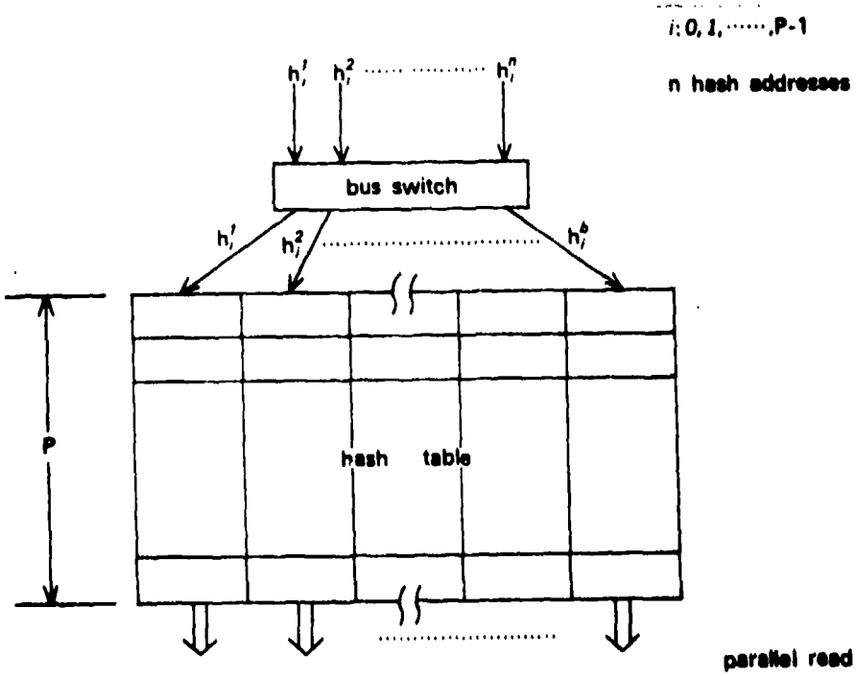


Figure 1 Parallel Hashing Scheme

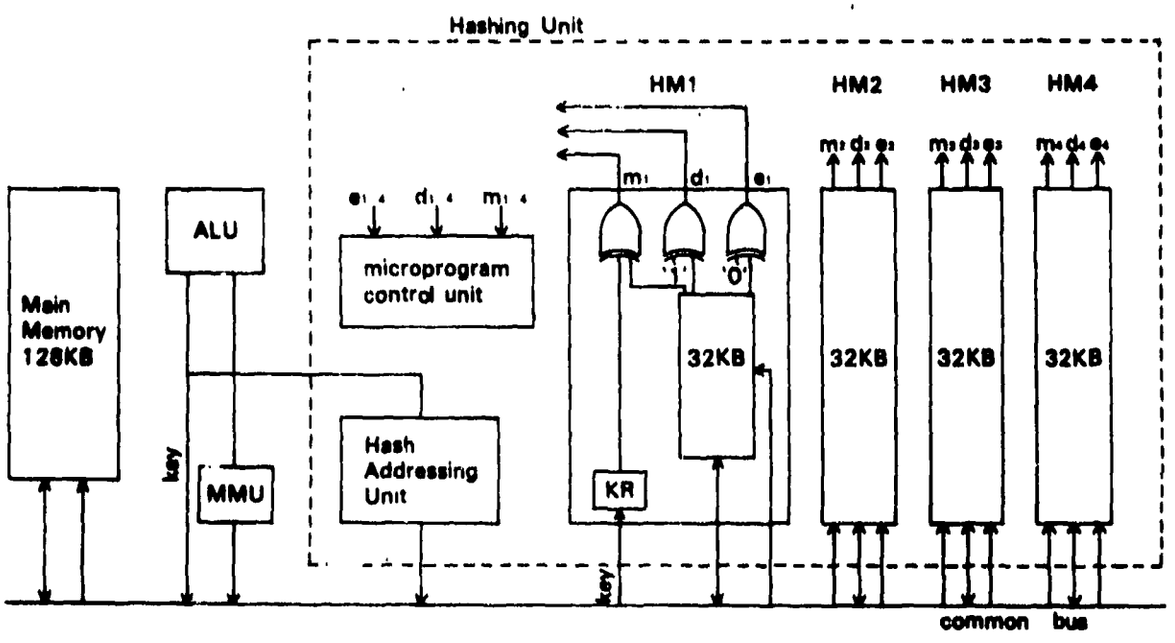


Figure 2 System with Hashing Hardware

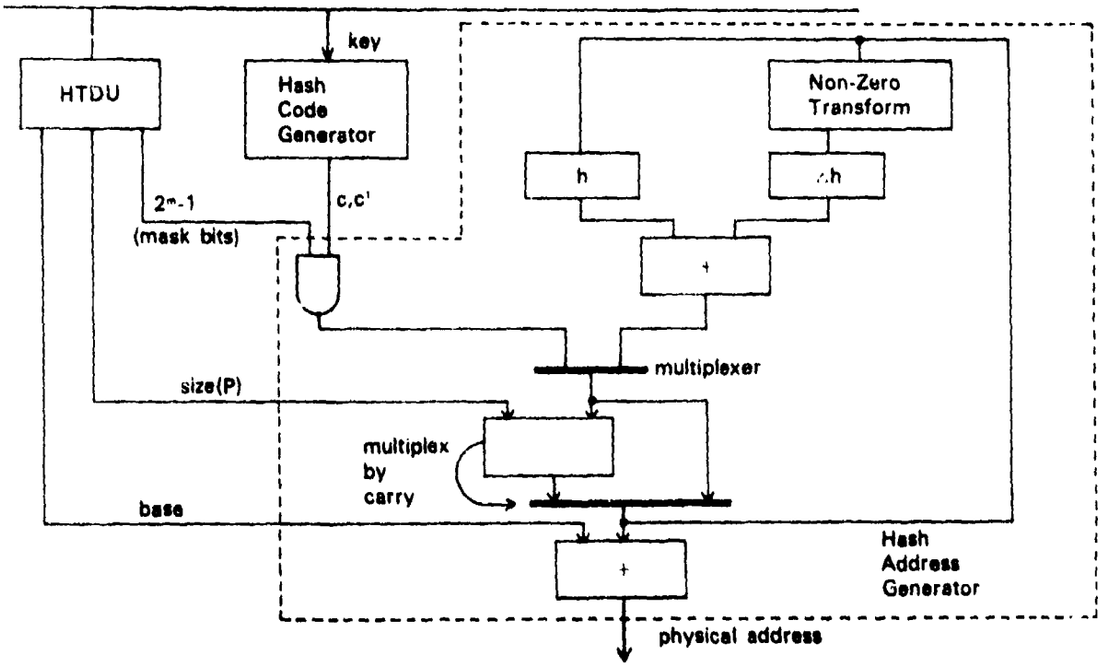


Figure 3 Block Diagram of Hash Addressing Unit

timing \ unit	HTDU	HCG	HAG	HM	microprogram control	ALU	time in total
step 1: t_0	select a hash table						t_0
step 2: t_0					check types		$2t_c$
step 3: t_0		generate hash code		transfer key to KR's			$3t_c$
step 4: t_1			compute h_0+base	occupy common bus cycle			$3t_0+t_c$
step 5: t_d			compute $\cdot h$	occupy common bus cycle (parallel read)			$3t_0+t_c+t_d$
step 6: t_0			compute $h_0=h_1+h$ h_1+base		multiple-jump on E, M	GR$_0$ matched address	$(3\cdot i)t_0+t_c+it_d$
step 7: t_d				occupy common bus cycle (parallel read)	jump to step 6		$(3\cdot i)t_0+t_c+(i+1)t_d$
step 8: t_0					jump to fetch routine	Set condition code	$(4\cdot i)t_0+t_c+it_d$

Figure 4 Control Sequence of HSR Execution

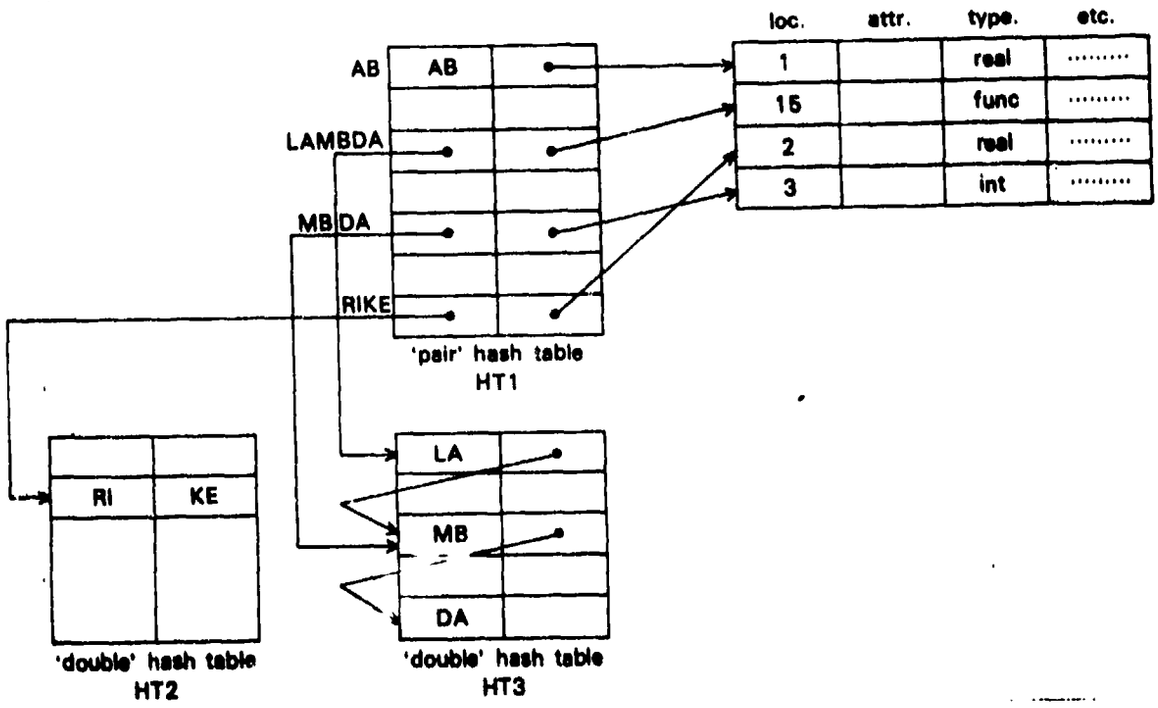


Figure 5 Representations of a Symbol Table

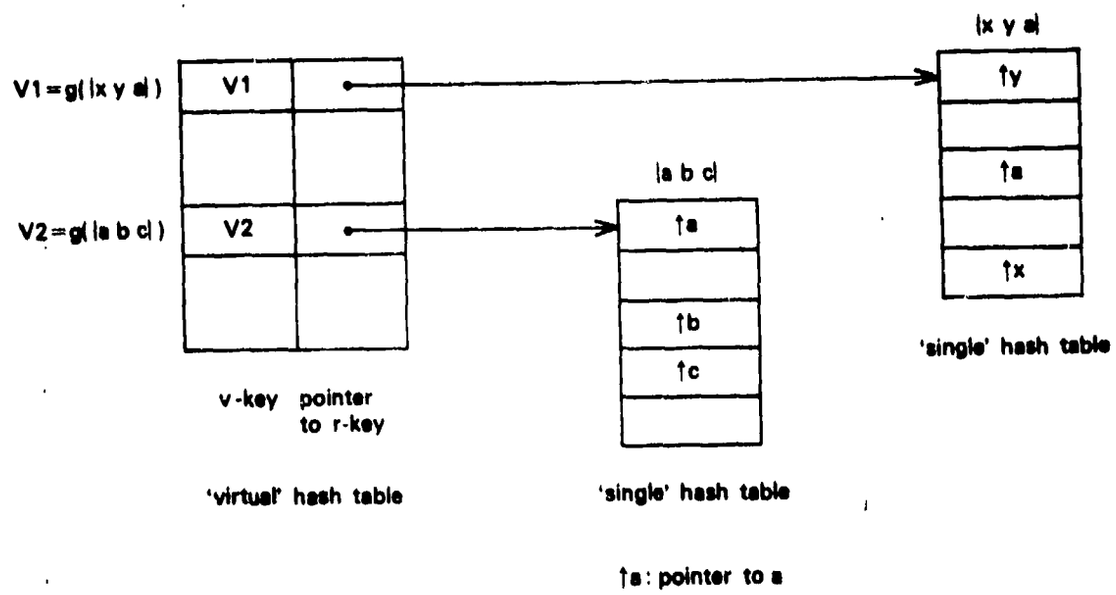


Figure 6 Representation of Sets Using Hash Tables

RAP.3 - A MULTI-MICROPROCESSOR CELL ARCHITECTURE FOR THE RAP DATABASE MACHINE

K.OFLAZER E.A.OZKARAHAN
Middle East Technical University, Ankara, Turkey

and

K.C.SMITH
University of Toronto, Toronto,Canada

Abstract

Recently introduced database machine proposals are critically reviewed. A new architecture for the cell processor of the RAP database machine utilizing multiple microprocessors and LSI serial memories is presented. The proposed cell processor designed down to the logic gate level, embodies concepts of modularity, flexibility, and firmware driven query processing. The concept of firmware execution of high level RAP assembler instructions is presented. The results of various analyses of the analytical and simulation models of the new architecture which were carried out elsewhere are summarized. Special emphasis is given to bulk memories that have the start-stop controllability (like magnetic bubble memories or RAM arrays simulating serial access) together with the increases in functional capability and performance obtained by incorporating such memories.

KEYWORDS: DATABASE MACHINES,ASSOCIATIVE PROCESSORS, DATABASE MANAGEMENT, LSI MEMORIES, MICROPROCESSORS, COMPUTER ARCHITECTURE

Introduction

The idea of providing backend computers for the efficient management of large databases, as a substitute for the slow software access methods, has received considerable attention in the recent years. The research efforts spent on this area have got the deserved recognition with the two special issues of IEEE journals[1,2]

In the last years, many specialized processors for handling the database management operations have been proposed. Among these there are CASSM[3] to process hierarchies and tables, RARES[4] for relational database management and RAP[5,6] that has been implemented at the University of Toronto and has also undergone certain changes[7]. DIRECT[8,9] is being implemented at the University of Wisconsin. Other proposals include the Database Computer (DBC)[10,11] as a backend processor-memory complex and the Bubble Memory Relational System[12].

In this paper we will first survey the most recent research efforts in the database machine field and then present a new approach to the RAP processor architecture, beyond that of RAP.2[7], utilizing LSI technology, like off-the shelf microprocessors, magnetic bubble memories (MBM), high density bulk RAM chips, etc.

Survey of Recent DBM Proposals

Most of the recent database machine proposals have exploited the advances in technology by incorporating microprocessors, CCD's, MBM's and the like.

DIRECT is a system for supporting relational databases. The system comprises a host for interfacing with the users, a backend controller for coordinating the overall database machine hardware and software, mass storage units for storing the database,a set of query processors, and CCD page frames for holding the relation pages that are being processed.

In this system, the query processors and CCD page frames are connected to each other by utilizing a cross-bar switch, so that all processors can access all page frames. Although this cross bar switch is much simpler than the conventional cross-bar switches, it may not be cost effective and may also reduce performance in larger implementations of this system as proposed in[9] with 10^3 processors. This is because, as the number of processors and page frames increases, the selector/decoder networks at the processor interfaces and the gating networks at the page frame interfaces of the cross-bar switch grow in size,thereby introducing extra delays in the data transfers between the processors and the page frames, and hence decreasing performance considerably.

Another feature of the DIRECT system is that the results of the basic relational algebra operations executed by the query processors are treated as temporary relations and are written onto free page frames allocated by the controller. The number of temporary relation page frames depends on the number of query processors assigned to the query.

This scheme increases the query processor-controller interaction during page frame processing because of temporary page frame requests and may introduce unnecessary page faults for some other set of query processors executing another query concurrently, just because their page frames may

be assigned to the temporary relations of a higher priority query. In this way, the degree of parallelism may drop seriously because of the creation of temporary relations. The temporary relations may cause a more serious performance degradation during the join operations in which the system page frame resources have to be partitioned for the source and result relations. The join operation may produce result relations with sizes comparable to the source relation and it is very likely that this system will suffer the thrashing problem in the join operation.

The Database Computer (DBC) is a system proposed for very large databases and a variety of data models, utilizing modified conventional moving head disks. The basic system comprises two processing loops; the structure loop for pipelined processing of the keywords and record indices and the data loop for actually processing the database contents.

One of the major drawbacks of this system is its way of representing data as attribute-value pairs. This scheme of repeating the attribute information wastes a considerable amount of data space. Another drawback is that the number of processors for doing the actual processing is very small compared with the database size; thereby reducing the parallelism that should be inherent in database machine systems. Furthermore, the number of interconnections required between the disk drive array and the track information processors may be prohibitive in terms of cost and physical requirements for the configuration proposed.

The DBC relies on the concept of partitioned content addressable memory (PCAM) for data accesses. A PCAM is one cylinder of a disk volume and is the largest amount of memory that can be processed with the limited amount of processors. One PCAM can be processed in one disk revolution, but if the qualification for a retrieval is complex and/or if the data to be processed occupies a large number of cylinders, then many disk revolutions are necessary for processing the data. The relational operation of join is also executed in a very inefficient manner. First, all the qualified domain values of the source relation are retrieved and then for each source value, another retrieval instruction over the target relation is issued. This implies that the number of instructions executed by the track information processors depends directly on the number of source domain values.

The performance study of this system in supporting relational databases[11] shows that a general purpose conventional computer performs better than DBC for large relations (e.g. with 20000 tuples) with reasonably large tuple sizes. This in turn implies that this system, although designed to support large data bases efficiently, cannot support a database with large relations as efficiently as a conventional computer despite the additional hardware costs introduced.

Furthermore, since this system relies also on the concept of index processing (although in hardware), the similar problems incurred by the update operations on conventional systems is likely to occur in DBC, because the structure memory should be updated as to reflect the result of the update.

Utilization of MBM's for supporting relational databases has been recently proposed by Chang[12]. The proposed hardware comprises MBM chips with certain augmentations to facilitate associative selections. A relation is mapped on one or more MBM chips with tuples across the minor loops and the domains along the minor loops. It is claimed by the author that augmentation of the MBM chips with off-chip indexing loops provides convenient indexing during data qualification and avoids redundant traversing of disqualified data. Two off-chip registers and a one bit comparator are provided for the database operations. The instruction set of this system is said to be inspired from that of RAP with minor changes.

The operational deficiencies of this system result from mainly the following: Since the hardware employed is substantially small and simple, provisions for in-place updates have not been provided. Furthermore, the existence of only one comparator limits parallel comparisons on data, hence limits query complexity. Also, the join operation is handled implicitly as in RAP, but only a single domain value from a source relation is transmitted to the target relation per scan. This mode of operation may severely degrade the performance of such a system in a join operation.

The following sections describe a restructuring of the RAP cell processor utilizing off-the-shelfmicroprocessors and bulk serial memories, especially MBM's. The proposed system differs considerably from the previous designs of RAP. First, the hardware structure of the cell is configured into a more regular and modular structure and the hardware complexity in terms of chip count has been reduced to a third of the previous designs. Secondly, query processing driven by microprocessor firmware and utilization of start/stop controllable memories such as MBM and/or high density RAM's permit highly complex data qualifications and highly efficient join operation. The proposed system can be considered as a RAP.3 system described in[7]. The reader, after following the paper, can draw a comparison of other database machines with the enhanced features of RAP, as also summarized in the conclusion, including especially the join operation.

. The RAP database machine can also be regarded as a good example of a High Level Language Computer Architecture. Since the context of the present discussion will deal with the architectural aspects of the new version of the RAP cell structure and the fact that the basic RAP architecture along with its instruction set are covered elsewhere[5,6,7], we will be content with providing only a summary description of the latest RAP instruction set in Appendix-4.

The New RAP Cell Processor Architecture

The new architectural approach having been adopted[16,17,18] restructures the RAP cell in yet another array of independently operating subcells, where each subcell comprises a microprocessor with necessary peripheral chips. Each subcell has the functional capability for the evaluation of complex data qualifications and in-place updates. To increase the effective functional speed of the cell, such subcells are duplicated in a linear array and a data flow strategy is incorporated to allow for parallel processing of consecutive tuples (i.e. relational record type occurrences) of a RAP relation[6,7], stored in the circulating memory (CM) of the cell. In this way, the overall RAP cellular system[6,7] becomes a system exploiting a two-dimensional parallelism within a linear array. Figure-1 shows the overall structure of the new RAP cell.

The CM of each cell also has a different structure than its counterpart in the previous designs[6,7]. The CM is chosen as word serial organization mainly to fit data access port size to the subcell microprocessor data bus width and also to incorporate rather slow, newly emerging bulk memory technologies like MBM's or high density RAM's (e.g. 64 K) in a parallel organization so as to enhance the effective data rate. A RAP relation is mapped directly onto the CM, so that the logical and physical structures of data are exactly the same. The format information required by the format sensing circuitry of the previous designs[6] is eliminated completely and the number of mark bit domains is increased to 16.

In data, numeric domains can be 2 or 4 bytes with 2's complement representation and non-numeric domains can be as long as required, provided that the sum of the domain lengths is less than or equal

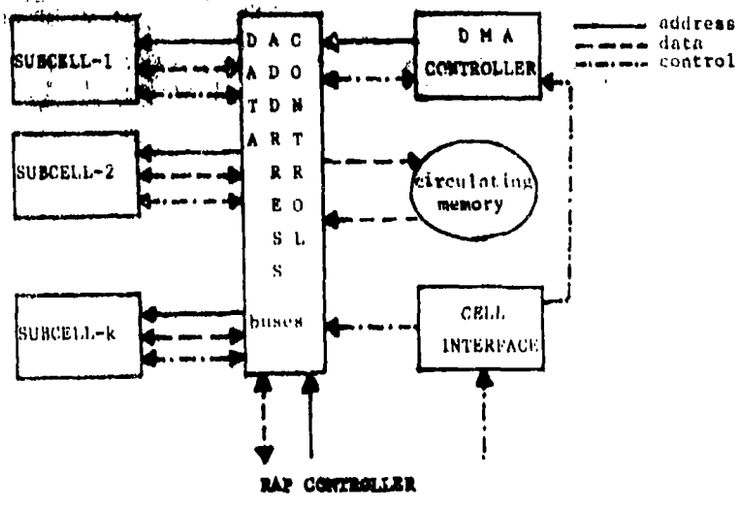
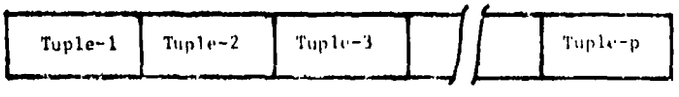
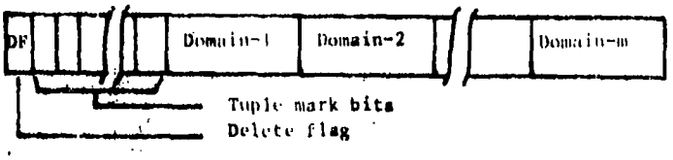


Figure-1 Structure of the new RAP cell.



a) Circulating memory structure



b) Tuple structure

Figure-2 Cell circulating memory format.

to the maximum tuple size of 1024 bytes. Furthermore, other data types like floating point numbers can be easily supported without any extra hardware. Figure-2 shows the format of the cell CM.

Operation of the Cell

The linear array of subcells provides multiple buffers (as small RAM's) for the tuples coming from the CM. At any time during CM circulation, more than one tuple can be out of the CM, which may be in the states of being loaded into a subcell buffer, being stored into CM from a subcell buffer, or being processed in a subcell. The existence of multiple buffers provides the necessary time for processing the tuples, thereby synchronizing the data move and data processing rates. The sequence of operations during a circulation of CM can be described with a process/time-slot diagram given in Figure-3.

In Figure 3, L_i, P_i and S_i, denote the load, process, and store states of some tuple for subcell$_i$, respectively. When the CM circulation is initiated successive tuples are loaded, via DMA, into successive subcells starting with subcell$_1$, until the end of (k-1)th tuple. In order to stay in synchronization, the first tuple should be stored from subcell$_1$, while the k th tuple is being loaded into subcell$_k$, and the 2nd tuple should be stored from subcell$_2$ while the (k+1)th tuple is being loaded into subcell$_1$, etc. During the circulation, each subcell microprocessor is initiated for processing as soon as its buffer is loaded with a new tuple.

It is evident that during the processing of CM contents, only (k-2) of k subcells are actually active at a given time. This may bring the idea of multiplexing (k-2) processors among k tuple buffers or, in general, multiplexing P processors among M tuple buffers where M>P. If M is not an integral multiple of P, then a general interconnection network (e.g. a cross-bar) should be utilized to allocate processors to buffers. If however M is an integral multiple of P, then a simple but static interconnection scheme for multiplexing each processor among (M/P) buffers may suffice. However in both cases, besides the interconnection complexity introduced, the important feature of CM wait time utilization (to be described later) cannot be made possible.

After pointing out this alternative to the original k-parallel microprocessor approach, the

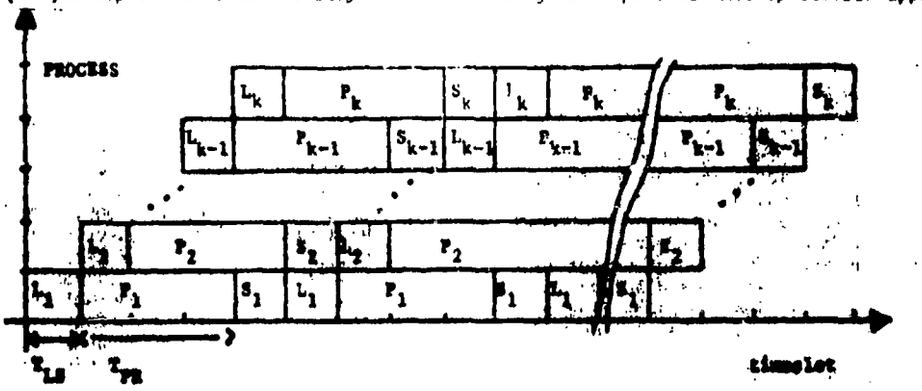


Figure - 3 : Load/Process/Store sequences of cell operation

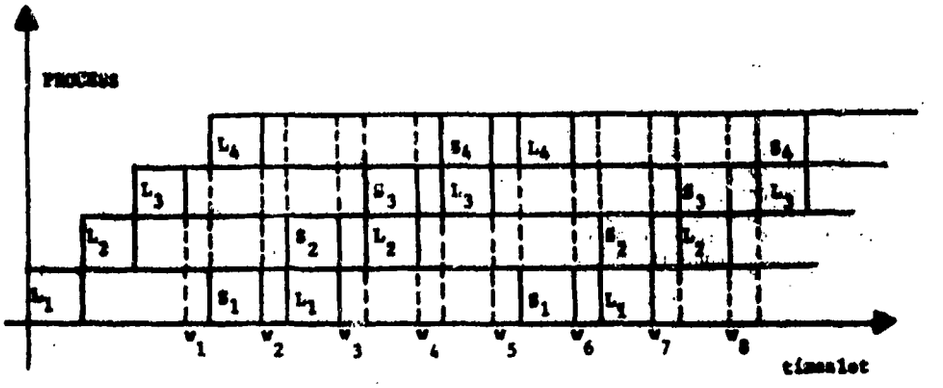


Figure - 4 : Illustration of MBM wait states for k::4

paper will continue dealing with the dedicated k parallel microprocessor approach to elaborate on the wait schemes and to preserve the modularity of the cell architecture.

As it was pointed out in [16], the processing time allocated for a subcell after its tuple is loaded is

$$T_{PR} = (k-2)*T_{LS}$$

where k is the number of subcells in a cell and T_{LS} is the DMA load/store time for a tuple. It should be noted that the allocated time depends on the tuple size and is larger for longer tuple sizes. In any case, the worst case expected processing time should be less than or equal to T_{PR} for a given tuple size so that synchronization is not lost. This constraint puts very high demands on the subcell microprocessor performance and on the number of subcells k (increasing k increases the allocated time) if the CM cannot be controlled in a start/stop fashion (as would be the case with rotating devices or CCD memories). Furthermore, this constraint limits the functional capability of the subcell by restricting the complexity of query qualification expressions.

The proper use of the start/stop feature of MBM's (or asynchronous access feature of bulk RAM's) relieves the above constraints, so that hardware parameters can stay within feasible limits. This is allowed in such a way that no performance degradation for average processing times occurs, while longer processing times corresponding to more complex qualification expressions impose a certain dynamic performance degradation which can be traded off with the issue of minimizing hardware. Furthermore, it is observed that in the execution of the relational join operation, handled implicitly in RAP[5] where a target relation (domain) value is matched disjunctively against an array of source relation (domain) values, the deliberate imposition of waits on CM (by stopping CM whenever necessary) reduces the overall time to execute the join operation. This point will be detailed in a following section.

Figure-4 shows the process/time-slot distribution for a controllable CM and for k = 4. The basic idea behind the utility of the start/stop feature of controllable memories can be stated in the following way: when the time comes to store a tuple from a subcell buffer (e.g. storing $subcell_1$ while loading $subcell_k$) if that subcell has not yet asserted that the processing of the tuple is complete, the CM is put temporarily in a wait state to allow for the completion of processing. The extra time requested by a subcell becomes also available to (k - 2) succeeding subcells so that the chance that they will impose further waits is highly reduced. An analysis of the timing of operations for this case is presented in Appendix 1.

Functions of the Basic Hardware Modules

The hardware modules given in Figure-1 have the following functions:

SUBCELLs: They process the tuples loaded into their buffers by the DMA CONTROLLER. The processing is driven by a query routine loaded into SUBCELL memories prior to the initiation of a RAP instruction.

DMA CONTROLLER: This module controls the simultaneous bidirectional data transfers between the cell memory and subcell buffers during the load/store operations. It also sequences the load/process/store operations and keeps track of the cell CM status.

BUSES: There are four buses that provide data, address and control paths between the cell modules during data transfers.

CELL INTERFACE: This module coordinates the overall cell operation during instruction initiation and termination, keeps track of cell status, and provides for the communication of the cell with the RAP array controller.

Query Execution

In the new architecture, the microprocessors of the subcells in each cell are the basic data processing units. Therefore, these microprocessors can be programmed to execute RAP instructions[5,19,20,21].

The basic idea behind the emulation of RAP instructions with microprocessor routines is that each RAP instruction can be mapped into what is called a "query routine". The basic RAP instruction constructs (i.e. MARK, RESET, MKED, UNMKED, updates, set-function computations, comparisons etc.[6]) have simple microprocessor code equivalents. Furthermore, the combination of the results of various qualification tests as disjunctions or conjunctions (or mixed which was not available in the previous designs) can be embedded into the sequential logic of the microprocessor query routine. This mapping brings considerable enhancements to RAP capabilities, since now, qualification complexities are limited only by the subcell microprocessor program memory size instead of the static hardware registers of the previous designs. Furthermore, since the whole tuple can be accessed during processing, domain to domain comparisons and updates are also made possible. An example of a query routine is provided in Appendix 3.

The subcell microprocessor memory comprises two parts. The ROM part contains the basic qualification evaluation routines (i.e. numeric and non-numeric value comparisons) and routines for the relational join and free variable operations. The RAM part is logically partitioned into two parts: one for the query routines and communication buffers, and the other for the tuple to be processed.

Before the initiation of a RAP instruction, the equivalent query routine and/or necessary parameters are loaded into the RAM's of the subcells of all the cells involved in the instruction, after the cell interfaces connect their cell

buses to the buses of the RAP controller.

Each time a CM circulation is started and whenever a new tuple is loaded into a subcell buffer, the microprocessor is forced out of the idle state to branch to the query routine. At the end of processing, a hardware flag is asserted to signal the DMA CONTROLLER so that the tuple can be stored back.

The cell interface is also controlled by a microprocessor, which after each RAP instruction is executed on the CM contents, polls each subcell and updates the cell status and computes (if applicable) cell set function subresults.

Execution of the Implicit Join Operation

The important and frequently encountered database operation of join, is done implicitly in RAP[5,21] by the cross-mark type commands. This operation is accomplished by extracting the qualified source domain values from the source relation cells and transmitting them to the target relation cells until all source (master) relation cells are processed. The execution of this operation had to be made as efficient as possible, because it was practically the only case where the superiority of the RAP system to conventional systems was estimated as to be less than 10-fold[13].

The new architecture employs a similar scheme for this operation. The values from qualified tuples of the first source relation cell are read out and buffered at the RAP controller, then a block of source values are loaded into target relation cell subcells and these cells are initiated for processing. This block loading is repeated until all of the buffered source values are processed; then the next source relation cell values are buffered and the above operations are repeated until all source relation cells are processed.

The number of source values loaded into target relation cells per circulation depends on the size of RAM space of the subcell, and in the current design, 400 2-byte numeric domain values (equivalently 200 4-byte numeric and a total of 800 bytes of non-numeric domain values) can be loaded and matched against a single target value. This number compared with 3 to 5 of previous RAP designs shows a significant improvement in the execution of the join operation. (The improvement however is not as much as the ratio of the loading factors due to the differences in the architectures and the fact that the cross-mark operation is now broken into discrete steps each starting at a new revolution (i.e. a repeated MARK instruction)). A snapshot of cross-mark execution is provided in Figure-5.

It is evident that processing that many source values imposes waits on the CM and hence increases the overall circulation time. However, it is observed that (in Appendix-2), if n is the number of source values that can be processed without imposing any waits, loading m x n (m >1) source values per circulation will reduce the number of circulations by (1/m) while the increase in each circulation time of the target relation cells will be significantly less than m-fold, because of the parallelism in the cell. In this way, the overall time to process a source cell with m x n values loaded per circulation will be less than the overall time with n values loaded per circulation.

Features of the New Design

The new RAP cell processor based upon the

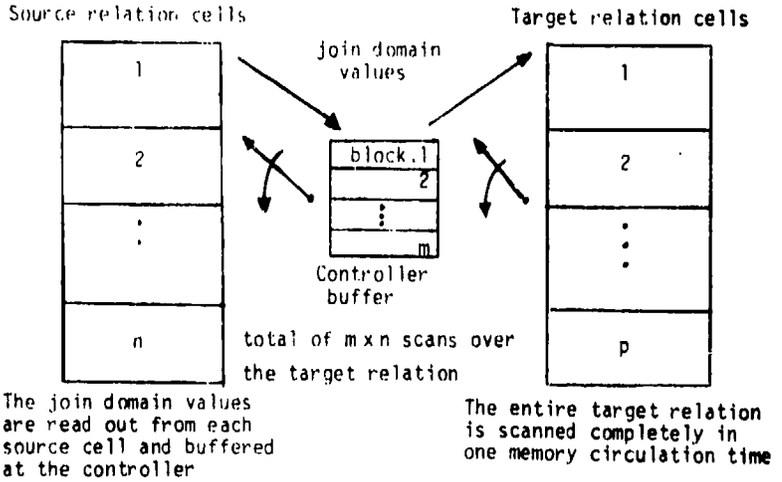
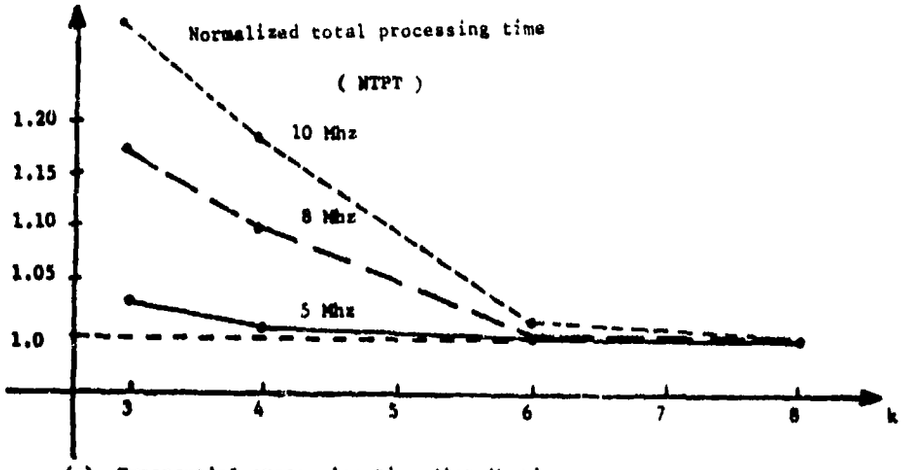


Figure-5 Execution of the cross-mark instruction

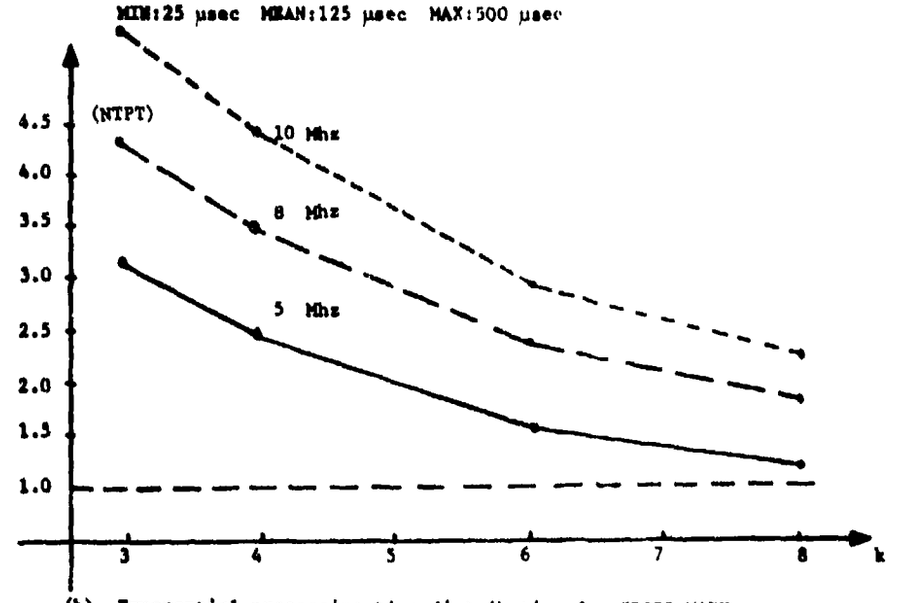
concepts presented above has been designed down to the gate level, together with the necessary microprocessor query routines for the general RAP instruction constructs[18].

In order to arrive at a decision for the number of subcells to use, various simulation studies were carried out[17,18]. Tuple processing times were sampled from two exponential distributions. The first distribution modeled processing times as to have a minimum of 25 μ sec, a mean of 125 μ secs and a maximum of 500 μ secs to reflect the average case. The second distribution had a mean of 1000 μ sec with 125 μ secs and 2000 μ secs as the bounds to model heavily loaded processing sessions as would be in a join operation. It was further assumed that the controllable memory array (16 bit wide) could deliver data with up to a 600 K Words/sec rate. The results of these experiments are provided in Figure-6.

It was decided that k = 4 would be a cost-effective choice to reduce hardware complexity



(a) Exponential processing time distribution
MIN:25 μsec MEAN:125 μsec MAX:500 μsec



(b) Exponential processing time distribution for CROSS_MARK
MIN:125 μsec MEAN:1000 μsec MAX:2000 usec

Figure - 6 :Plot of normalized processing time vs k
(data rate as parameter)

and impose practically no waits for the average processing times at the memory rate of 300 K Words/sec (5 M bits/sec)which is attainable by the current MBM's.

The cell design utilizes 4 subcells where each subcell contains an Intel-8086 microprocessor with 2 K bytes of RAM and 1 K bytes of ROM and some additional control logic. Total chip count per subcell is 20. The cell memory interface is configured for 16 x 92 K bit MBM's but can easily be modified for other types of MBM's and/or bulk RAM's (The reader, although not implied in the paper, should not be disillusioned by the fact that other types of bulk serial or block addressable memories cannot be supported. They can be with the exception of not having the further performance gains achievable by the controllability feature. The architecture could also be conceptualized as having a bulk RAM memory with a single microprocessor similar to the original design. However, the speed to be imposed on a single microprocessor will be beyond those conjectured for the future at least at the cost effective scales. Cost of RAM's would be another issue which must be cheap and competitive despite their volatility). The total chip count of this configuration is 160 per cell which is slightly over one third of that of the previous designs.

It should be emphasized that utilization of 8086's is a specific case of the implementation of the proposed architecture. In fact, besides the large data bandwidth, only the powerful string operation instructions and a suitable subset of the remaining general purpose instructions of the 8086 are utilized for implementing the subcell firmware. In a possible large scale commercial implementation, a special purpose microprocessor with only the necessary instructions can be developed and utilized. Depending on the cost versus speed trade-offs, it is also possible to implement the proposed architecture with powerful 8-bit microprocessors having fast block operations.

In memory, the CM data rates can be as high as technology permits. For example, the 8086 based system can support a 16 M bit/sec burst data rate for low to medium complexity qualification terms of RAP instructions without any serious performance degradation due to the utilization of waits. It may be concluded that, it is the limitations of controllable memories (e.g. MBM's) that will be the determining factor for the terminal speed of the proposed architecture.

The simulation studies and analytical modeling of the cell operation show that considerable performance improvements over previous RAP designs can be attained. It has been observed by simulation[18] that the new processor performs 3-6 times better than the previous designs despite the fact that a larger and slower memory is being incorporated.

The join operation, which has not been emphasized (from a performance point of view) in other database machines, can be performed rather efficiently, because a larger number of values can be matched during each circulation.

Furthermore, since all the cell status information is kept by a microprocessor at the cell interface, task switching in a preemptive resume multiprogramming environment[5,15], requires no extra hardware. Relation status saving and restoring are accomplished by the two new RAP instructions SAVE-MARKS and RESTORE-MARKS[19,20] which save and restore tuple mark bits into and from special domains appended to the end of each tuple that serve as a push down stack during task switchings.

The overall RAP system configuration with the new processor architecture would be similar to previous RAP configurations[6,7], only that the controller for the cell array, which is currently being designed, is expected to be a more intelligent unit. Its main functions will to be to keep track of device status by maintaining necessary relation and cell status tables, instruction scheduling for a RAP query whose instructions have been converted to microprocessor code, data buffering in join operations, control of hardware and software iterative instructions, computation of overall set function results and communication with the frontend computer. It is also expected to do the functions of the monitor[15] for the RAP multiprogramming and virtual memory operations. The entire cell-array controller configuration will be driven by a conventional frontend computer to interface the users.

Conclusion

After a survey of recent database machine proposals, a new architecture for the RAP database machine's cell processor is presented. The new architecture has certain advantages over the previous hardwired RAP designs. Mainly, the hardware complexity is decreased while the operational flexibility is increased. The utilization of LSI components opens the way for the modularity of the architecture. The utilization of controllable memories also relieves the architecture from the constraints of worst case timing requirements.

From a feature comparison point of view the proposed architecture has the following properties one or more of which are not shared by the other database machines:

a) Data qualifications of any complexity can be evaluated over the memory contents in one circulation of the memory.

b) All kinds of updates and arithmetic operations can be done on the memory contents without transfering data in and out of the RAP system.

c) Join operation is handled in a very efficient manner. In most of the typical cases, one target relation cell memory circulation may suffice to process the values of one source

relation cell, compared to the large number of circulations (or revolutions) required in the other database machine proposals.

d) Since no software access methods are utilized, no overhead on the frontend computer is imposed.

e) A multiprogramming environment can be attained without any extra hardware.

f) It is expected that a single RAP database machine is going to be confined within certain practical physical limits. In order to support very large database applications, either one or combination of the following two system configurations can be incorporated:

1) Virtual memory back up as in[5,14] for a single processor

2) The database can be distributed in a network of RAP database machines and a given database operation can be decomposed and executed on the network of modest size RAP's concurrently, as shown by a previous study[22].

RAP.3 prototype implementation, along with its already operational software, is nearing completion at the METU.

Acknowledgement

We gratefully thank Intel Corporation for the donation of 8086 microprocessors and memory chips.

Appendix 1

Timing Analysis of Cell Operations

The following analysis describes the relationships among certain timing parameters.

Let

T_{BIT} ≡ CM bit time = CM shift time/16

TUPLEN ≡ length of a tuple in bits

k ≡ number of subcells/cell (k>3 because of the data move strategy incorporated)

T_{LS} ≡ time to load (store) a tuple via DMA = T_{BIT}*TUPLEN

T_{AVL_i} ≡ available time to process tuple i

NT ≡ number of tuples in CM

T_{TOTAL} ≡ total circulation time of CM from the start of loading of the first tuple to the end of storing of the last tuple.

T_{REST} ≡ extra time needed to restore the last (k-1) tuples. (It should be noted that CM circulation is completed only after the last tuple is restored. Some extra time is needed to restore the last (k-1) tuples because the total dynamic capacity of the cell memory is equal to the CM capacity plus the capacities of the (k-1) subcell buffers).

T_{WAIT} ≡ total time during which the cell memory is in the wait state in a circulation.

Then we have the following relationships:

$$T_{AVL_i} = (k-2)*T_{LS} + \sum_{\substack{j=i-(k-2) \\ j \geq k-2}}^{i-1} w_j ; (i=1,\ldots,NT \text{ and } w_0=0)$$

where w_j are the wait times associated with $tuple_j$ (ref.Figure 4).

$$T_{WAIT} = \sum_{j=1}^{NT} w_j$$

$$T_{REST} = (k-1)\ T_{LS}$$

$$T_{TOTAL} = NT * T_{LS} + T_{WAIT} + T_{REST}$$

$$= (NT+k-1) * T_{LS} + T_{WAIT}$$

It should be noted that T_{WAIT} is dependent on the complexity of the query routine (if k and T_{BIT} are fixed), but an upper bound on T_{TOTAL} can we derived as follows:

Assume that all tuples require exactly L times the time allowed by the architecture i.e.:

$$T_{REQ} = L*(k-2)*T_{LS} \ ; \ (L \geq 1)$$

Assuming also that mod (NT,k) = 0, then during the circulation, NT/k tuples will be processed by each subcell. The time to handle a tuple is:

$$T_{TUPLE} = T_{REQ} + 2 * T_{LS}$$

where the last term accounts for the load and store times.

Since processing of the tuples are overlapped over the k subcells, the total time for a circulation will be:

$$T_{TOTAL} = (NT/k) * T_{TUPLE} + (k-1) * T_{LS} + (L-1) * (k-2) * T_{LS}$$

where the first term is the time to process NT tuples with k subcells in parallel, the second term is the time to restore the (k-1) tuples at the end of the circulation and the third term is the initial extra time (beyond the allocated time) required by $subcell_1$ for $tuple_1$. Inserting T_{TUPLE} gives on upper bound for the circulation time when each tuple requires L times the allocated time, as:

$$T_{TOTAL} = ((NT/k)*(2+L*(k-2)) + L*(k-2)+1) * T_{LS}$$

Appendix 2

Analysis of the Join Operation Performance

The time to process one source relation cell contents can be approximated as:

$$T_{JOIN} = T_{BUF} + \left|\frac{NS}{n}\right| * T_{TOTAL}$$

where the first term is the time to read and buffer the source cell values (1 CM circulation) and the second term is the time to process the NS buffered source values and represents $\lceil NS/n \rceil$ circulations (i.e., n values are passed in each circulation) of the target relation cells.

If n_1 is the number of source values that can be processed in one target relation cell memory circulation without imposing any waits ($L = 1$), then the total circulation time for this case will be (ref. Appendix 1):

$$T_{TOTAL,nowait} = (NT+k-1) * T_{LS}$$

while the total number of such circulations will be $\lceil NS/n_1 \rceil$.

If $m*n_1$ source values are processed in one target relation cell memory circulation, then L will be roughly m, and the total circulation time will be:

$$T_{TOTAL,\ wait} = ((NT/k)(2+m(k-2)) + m\ (k-2)+1) * T_{LS}$$

while the total number of such circulations will be $\lceil NS/(m*n_1) \rceil$.

Neglecting the terms k-1 and m(k-2)+1 in the last two equations, which are much less than the corresponding terms we can write:

$$\frac{T_{TOTAL,\ wait}}{T_{TOTAL,\ nowait}} = \frac{(NT/k)(2+m(k-2))}{NT} = \frac{2+m(k-2)}{k} < m \text{ for } m > 1$$

It can be observed that imposing waits on the CM by feeding in more source values per circulation reduce the number of circulations by a factor of 1/m while the increase in the total time of each such circulation is less than m-fold, hence the overall time to process the buffered source values is reduced.

The actual execution time in reality will be much less than the above derived bounds because of the fact that after each target relation scan, the number of target values not yet selected and hence will impose waits, will diminish at an increasing rate until the last target relation scan.

The current design employs four subcells (k=4) and assumes that CM shifts at 300 kHz giving a T_{BIT} of 208 nsec/bit; then for 1 Kbit target relation tuples, the allocated time is 426 μsecs. Within this time, the INTEL 8086 routine developed to perform equi-join on 2 byte numeric domains can process 100 source values without imposing any waits. Processing 400 such source values gives $L \approx 4$. Since a target relation value may qualify for the join before the whole source

```
QUERYRTN  :  LEA   BP,MASKD          / check if tuple is deleted
             CALL  MKED              / previously;
             JB    NOTQUAL           / exit if deleted;
             LEA   BP,MAKT4          / check if tuple is T4 marked   } MKED (T4)      ?
             CALL  MKED              / previously;                   }
             JNB   NOTQUAL           / exit if not T4 marked;
             LEA   BP,PB1            / set pointer to parameter block 1; }
             CALL  COMPNUM2          / call numeric comparison routine;  } SALARY   2000 ?
             JNB   NOTQUAL           / exit if comparison fails;         }
             LEA   BP,PB2            / set pointer to parameter block 2; }
             CALL  COMPLITR          / call literal comparison routine;  } DEPT = 'SHOE' ?
             JNB   NOTQUAL           / exit if comparison fails;         }
             LEA   BP,PB3            / set pointer to parameter block 3; }
             CALL  ADD2              / tuple is qualified, update it     } ADD 500 TO SALARY
NOTQUAL   :  JMP   WAIT              / and wait until next tuple;
MASKD     :  DC    X'8000'           / mask for deleted tuples;
MASKT4    :  DC    X'0800'           / T4 marked mask;
PB1       :  DC    A(TUPLE+SALARY)   / address of SALARY domain in buffer;
             DC    H'200'            / external  comparand;
             DC    H'4'              / comparison mode for "Greater than";
PB2       :  DC    A(TUPLE+DEPT)     / address of DEPT domain in buffer;
             DC    H'8'              / length of the domain;
             DC    H'2'              / comparison mode for "equal to";
             DC    C'SHOE'           / external comparand;
PB3       :  DC    A(TUPLE+SALARY)   /
             DC    H'500'            / external value to be added.
```

Figure-7 Intel 8086 Program for a RAP Instruction

value block is scanned, the actual average total circulation time will be considerably less than what was found in the above analysis for the worst-case assumptions.

Appendix 3

Query Routine Example

Consider the RAP instruction:

```
ADD [EMP (SALARY): MKED (T4) &
     SALARY > 2000 & DEPT = 'SHOE'] [500]
```

which adds 500 to the salaries of those employees which satisfy the accompanying qualification expression. It is assumed that SALARY is a 2 byte numeric domain and DEPT is an 8 byte literal domain.

The query routine for this RAP instruction, in INTEL 8086 instruction set, can be given as in Figure-7.

The routines MKED, COMPNUM2, COMPLITR and ADD2 reside in subcell ROM and perform mark status tests, value comparisons and addition updates on the domains of the tuples according to the information supplied with the associated parameter blocks. Since the data qualification evaluation and the update are done together, this instruction would take only one cell memory circulation to process all the tuples of a relation.

It should be noted that, it is possible to construct query routines for data qualifications and/or updates of any complexity.

Appendix 4

Summary of the instruction set of the RAP DBMS Assembler language

Selection and retrieval commands: Implement selection and/or data retrieval.

```
MARK            : Selects and tags
RESET           : Selects and removes tags
READ            : Selects and reads
CROSS_MARK      : Maps between two record types
CRS_COND_MARK   : Maps between two record types
GET_FIRST_MARK  : Cursor and mapping within a
                  record type
GET_FIRST       : Cursor
SAVE            : Selects and saves item in RAP
                  register
```

Update commands: Perform selection and in-place arithmetic and replacement updates.

```
ADD             : Item1 ← Item1 + Item2 (or constant)
SUB             : Item1 ← Item1 - Item2 (or constant)
MUL             : Item1 ← Item1 * Item2 (or constant)
DIV             : Item1 ← Item1 / Item2 (or constant)
REPLACE         : Item1 ← Item2
```

Statistical (Set function) commands: Select and compute functions in-place.

```
SUM             : Selects and accumulates
COUNT           : Selects and counts
MAX             : Selects and finds the maximum
MIN             : Selects and finds the minimum
AVERAGE         : Selects and computes average
```

Insertion and deletion commands: Insert and delete record occurrences.

```
DELETE          : Selects and deletes record
                  occurrences from the record type
INSERT          : Inserts record occurrences into
                  the record type
```

Data definition commands: Initialize, populate, and delete a record type.

```
RELATION        : Defines a new relation (record
                  type). Size, type, length
                  parameters for the data are
                  declared. (Key attributes and
                  access paths are defined if the
                  sofware emulator rather than the
                  actual machine is used). User
                  capabilities, access rights, and
                  the protection parameters are also
                  declared with the use of this
                  command.
CREATE          : Populates the database for the
                  specific record types which have been
                  defined by the RELATION command.
DESTROY         : Deletes a record type
```

System commands:

```
AUTHORIZE       : Grants access to the user via a
                  password
LOCK            : Specified record types are locked
                  against concurrent accesses
RELEASE         : Releases locks
SAVE_MARKS      : Current mark bits of specified
                  relations are pushed onto stacks of
                  each tuple
RESTORE_MARKS   : Restores marks by poping the
                  saved mark bits
LOCATE          : Returns the node address of the
                  relation being searched
MOVE            : Moves an entire or restricted
                  subset of a relation to the
                  specified site
STATUS          : Performs dynamic status checking
                  for branching purposes
READ_MARKS      : Same as READ, but output includes
                  also mark bits
```

Register manipulation commands:

```
READ_REG        : Reads out RAP registers
STORE_REG       : Enters data into user registers
DEC_REC         : Decrements specified register
                  contents by one
INC_REC         : Increments specified register
                  contents by one
RADD,RSUB,RMUL,RDIV: Perform specified arithmetic
                  operations on registers as:
                  <reg> ← <reg><ropr><operand> where
                  ropr is one of RADD,RSUB,RMUL, or
                  RDIV.
```

Decision and transfer commands: Control program loops.

TEST	: Tests presence of tags within a record type
BC	: Branch, conditional and unconditional
EOQ	: End-of-query

References

1) IEEE COMPUTER, Special Issue on Database Machines, Vol.12, No.3, March 1979.

2) IEEE Transactions on Computers, Special Issue on Database Machines, Vol.C-28, No.6, June 1979.

3) COPELAND, G.P.,LIPOVSKI, G.L.,SU, S.Y.W., "The architecture of CASSM: A cellular system for non-numeric processing", Proceedings of First Annual Symposium on Computer Architecture, 1973, pp.121-128.

4) LIN, C.S.,SMITH, D.C.P.,SMITH, J.M., "The design of a rotating associative memory for relational database applications", ACM Transactions on Database Systems, Vol.1, No.1, March 1976, pp.53-65.

5) OZKARAHAN, E.A., "An associative processor for relational databases-RAP", Ph.D.Thesis, Department of Computer Science, Univ. of Toronto, January 1976.

6) OZKARAHAN, E.A.,SCHUSTER, S.A.,SMITH, K.C., "RAP-An associative processor for database management", AFIPS, Proceedings of NCC, Vol. 44, 1975, pp.379-387.

7) SCHUSTER, S.A.,NYUGEN, H.B., OZKARAHAN, E.A.,SMITH, K.C., "RAP-2, An associative processor for databases and its application", Proceedings of 5 th Annual Symposium on Computer Architecture, Palo Alto, April 1978, pp.52-59. Also in the special issue, IEEE Transactions on Computers, Vol.C-28, No.6, June 1979.

8) DEWITT, D.J., "Direct-A multiprocessor organization for supporting relational database management systems", Proceedings of 5 th Annual Symposium on Computer Architecture, Palo Alto, April 1978, pp.182-189.

9) DEWITT, D.J., "Query Execution in Direct", ACM-SIGMOD Conference Proceedings, May 1979, pp.13-22.

10) HSIAO, D.K.,KANNAN, K., "The architecture of a database computer-A summary", Proceedings of 3 rd Workshop on Computer Architecture for Non-Numeric Processing, May 1977.

11) BANERJEE,J.,HSIAO, D.K., "Performance study of a database machine in supporting relational databases", Proceedings of 4 th Int.Conference on Very Large Databases, Berlin, September 1978, pp.319-329.

12) CHANG, H., "On bubble memories and relational database", Proceedings of 4 th Int. Conference on Very Large Databases, Berlin, September 1978, pp.207-229.

13) OZKARAHAN, E.A., SCHUSTER, S.A.,SEVCIK, K.C., "Performance evaluation of a relational associative processor", ACM Transactions on Database Systems, Vol.2, No.2, June 1977, pp.175-195.

14) SCHUSTER, S.A.,OZKARAHAN, E.A.,SMITH, K.C., "A virtual memory system for a relational associative processor", AFIPS, Proceedings of NCC, Vol.45, 1975, pp.291-296.

15) OZKARAHAN, E.A.,SEVCIK, K.C., "Analysis of architectural features for enhancing the performance of a database machine", ACM Transactions on Database Systems, Vol.2, No.4, December 1977, pp.297-316.

16) OZKARAHAN, E.A.,OFLAZER, K., "Microprocessor based modular database processors", Proceedings of the 4 th Int. Conference on Very Large Databases, Berlin, September 1978, pp.300-311.

17) OFLAZER, K.,OZKARAHAN, E.A., "A Multi-microprocessor architecture for a cellular database machine-RAP", Technical Report IS-DB.5, Dept. of Computer Engineering, METU, December 1978.

18) OFLAZER, K., "A Microprocessor based approach to RAP database machine cell structure-Design and Analysis", M.Sc. Thesis, Dept. of Computer Engineering, METU, June 1979.

19) ONLO, S., "Design and implementation of a software emulator for the Relational Associative Processor-RAP", M.Sc. Thesis, Dept. of Computer Engineering, METU, August 1979.

20) OZKARAHAN, E.A.,ONLO, S., "The revised RAP instruction set and the RAP software emulator", to appear.

21) OZKARAHAN, E.A.,SCHUSTER, S.A., "A High-level machine oriented query language for a Relational Associative Processor", Computer Systems Research Group Technical Report CSRG-74, University of Toronto, 1976.

22) TANSEL, A.U.,OZKARAHAN, E.A., "Query Execution in Distributed RAP Database Machine Systems", Dept. of Computer Engineering, Technical Report IS-DB-6, Middle East Technical University, 1978.

This work is supported in part by the NATO research number RG002.80 of the Scientific Affairs Division.

ARCHITECTURE OF A MULTI-LANGUAGE PROCESSOR BASED ON LIST-STRUCTURED DELs

J.P. SANSONNET
M. CASTAN
C. PERCEBOIS

Laboratoire "Langages et Systèmes Informatiques"
Université Paul Sabatier
118, route de Narbonne 31077 TOULOUSE CEDEX
FRANCE

ABSTRACT

A direct-execution model, based on the tree-structured internal representation of the source-texts has been defined. It features a single intermediate environment and two environment transfers : the first one corresponds to a bidirectional translation between the source-text and the tree-structured internal form. The second one is a conventional microprogrammed interpretative process on a specialized hardware architecture.

In this paper, a full description of a hardware architecture which directly holds the tree-structured forms is given. Its characteristic features are discussed and the micro-control operations which deal with the main tree-structured form concepts (recursivity, top-down tree traversing, escapes) are presented.

1 - INTRODUCTION

To solve the problems resulting from the semantic gap, which arise in the conventional computer systems, new computer architectures have been revealed these last few years. Their purpose is to support directly one or more high level languages, in hardware. In this way, eliminating the order-codes tends to close the gap between the high level language and the physical structure of the host machine.

Although the Von Neumann architecture is increasingly and rightly questioned none of the proposed systems of high level language processors have been traded successfully. We tried to analyse the reasons of these failures[1,2] and it appears that the attractiveness of the Von Neumann architecture resides in its conceptual simplicity, whereas the suggested solutions[3,4,5] are characterized by complex models, difficult to understand and to implement, and often leading to gas-works architectures.

Therefore, we have proposed a direct execution scheme, based upon the definition of a class of list-structured Directly Executable Languages (DELs), which is derived from LISP[6]. The objective of this scheme is to provide the implementation of high level languages with a systematic support, easy to understand, and to use[7].

1.1. The 3L-model

A direct execution scheme with a single level was defined i.e. a scheme including only one intermediate environment between the source-text and the executional environment (fig.1).

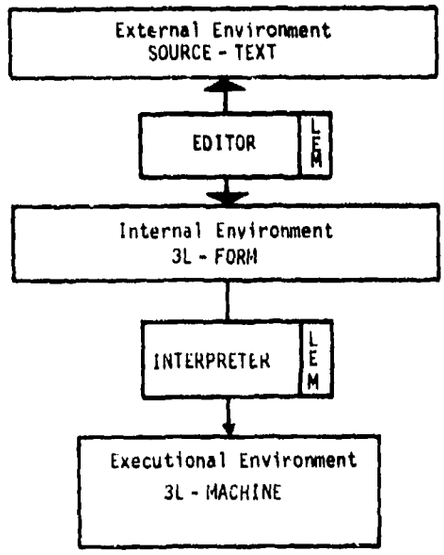


Fig.1 - The 3L-model

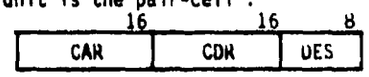
- A first interactive processor, the editor, is responsible for the communication between the external environment (source-text) and the internal environment (DEL).
- A second processor, the interpreter, is responsible for the evaluation of the internal form through the hardware operators.
- The 3L-machine (M3L) is the physical support of the 3L-model. Both processors are microprogrammed on M3L, with a high level microprogramming language, specialized in the expression of the emulation processing : the Language for EMulation (LEM).

1.2. The 3L-form

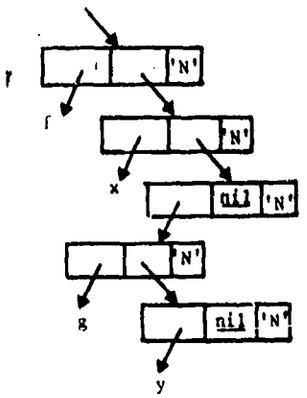
The choice of the intermediate environment determines the direct execution scheme. As we wished to

maintain the whole semantics of the source-text while providing the interpreter with an easy form to handle, we chose a list-structured internal form, based upon LISP : the LISP-Like-Languages (3L). The 3L form is prefixed and fully parenthetized. Although its semantic power is very high, its syntax is absolutely trivial and it offers a great systematization for the internal representation of the programs.

The 3L form is represented within the memory by a binary tree-structured form. This form is tagged, its unit is the pair-cell :



the CAR field generally represents a left pointer, the CDR field a right pointer, and the DES field gives the description of the cell content, more precisely for the representation of objects.



N : node

Example : Suppose that in the high level language we have the operation f(x,g(y)). It can be expressed in the terms of the symbolic 3L form as (f x (g y)), and within the pair-cell memory :

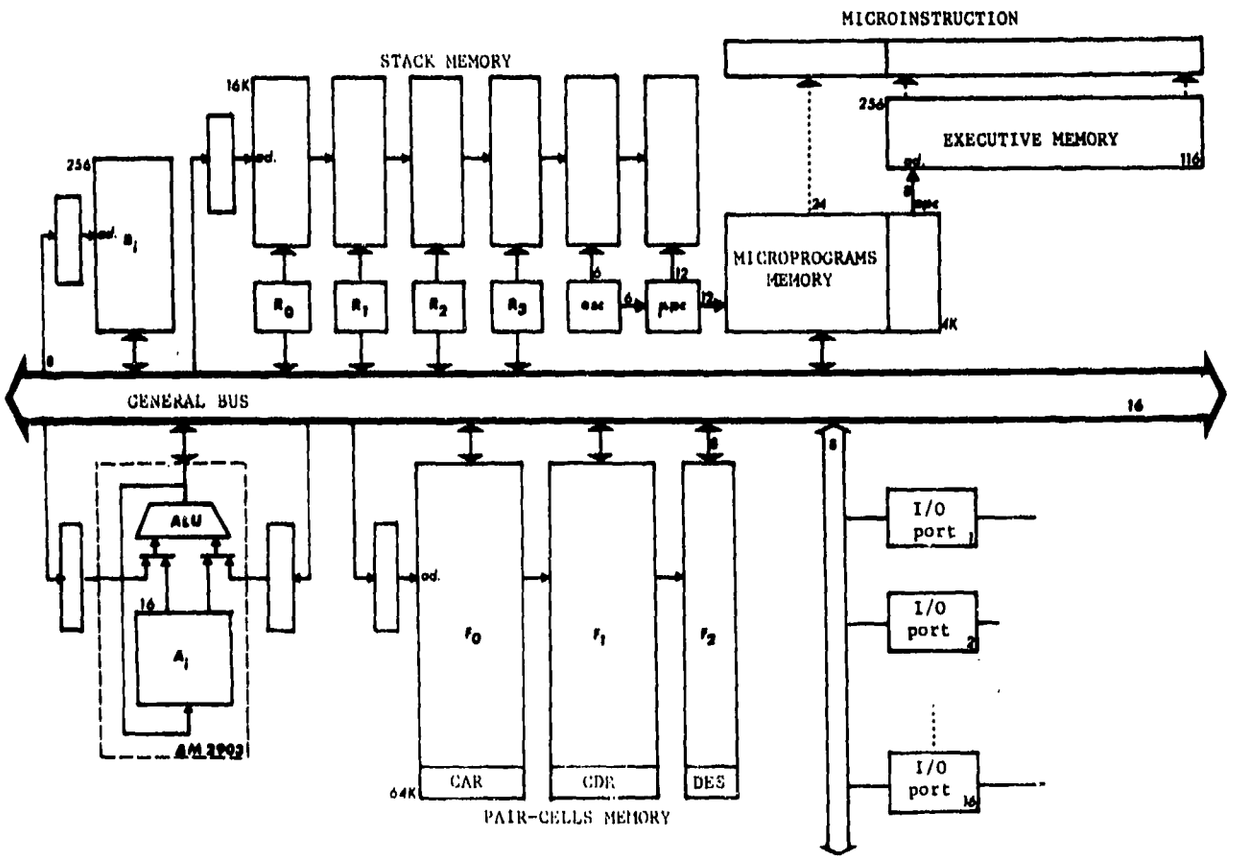


Fig. 2 - ARCHITECTURE OF M3L

2 - THE GENERAL STRUCTURE OF M3L

The M3L project started with a systematic study of the interpretation of LISP. First, we defined a pseudo-machine, then we wrote a simulator, and developed a microprogrammed LISP interpreter upon it. The simulation measures[1] opened up on a new architecture, which was defined for the M3L prototype, presently in the achievement phase.

2.1. Synoptic of M3L

The general organization of the 3L-machine is very simple. The resources are interconnected via a single bus which determines the datapath. The datapath is 16-bit wide, being the maximal size of the prototype pair-cells memory. (fig.2)

In the 3L-machine there are four categories of registers :

A_i registers $i \in [0,15]$
they are used for current works and information transfers between microprocedures

B_i registers $i \in [0,255]$
they serve as global registers for every microprocedure, they contain the descriptors of the current emulated system

Γ_i registers $i \in [0,31]$
they are flip-flops which give the status of the system. They are global resources and some of them can be set or reset by the programmer

R_i registers $i \in [0,3]$
they make the recursivity in LEM possible by the use of their locality.

2.2. The numerical processing

In the Von Neumann architecture, the numerical processing is prevalent. It is represented by the central operator and the inputs/outputs. More and more, it is integrated, especially in the microsystems. On the contrary, in a high level language processor the non-numerical processing is prevalent. It is true for M3L where the architecture is designed according to the emulation processing. Of course it is yet necessary to incorporate the elements of the numerical processing within this architecture. Nevertheless, they take a marginal place in M3L and they are entirely supported by a single LSI family (AMD 2900).

. The arithmetical processor

Most of the arithmetical functions of the 3L machine are performed by a monolithic processor (AM 9511). This processor relieves the machine of all the corresponding micro-software of mean importance for emulation. It can be viewed as a peripheral of M3L, it runs in parallel, and it is interfaced by the general bus. The main operations performed by the AM 9511 are :

- 18 data manipulation operations : conversions fixed-float, read, write, ...
- 5 fixed arithmetical operations (16 and 32 bits)
- 4 float arithmetical operations (32 bits) : +, -, *, /
- 11 secondary operations (32 bits float) : $\sqrt{}$, sin, cos, x^y, ...

. The arithmetical and logical unit (ALU)

The ALU of M3L is built from four AM 2903 LSI chips. Owing to the use of an arithmetical processor, its task is very small : it has to manage the A_i registers, and it performs data comparisons which are typical tasks of the environment transfers.

. Inputs/outputs

The inputs/outputs system is built from a 8 bit wide peripheral minibus on which the interface adaptators for asynchronal communications are connected. These chips perform the standard control functions according to the CCITT V24 Standard. The minimal version of M3L includes an ACIA for driving the TTY, and another for interacting with a microsystem, responsible for the management of inputs/outputs and disk-files.

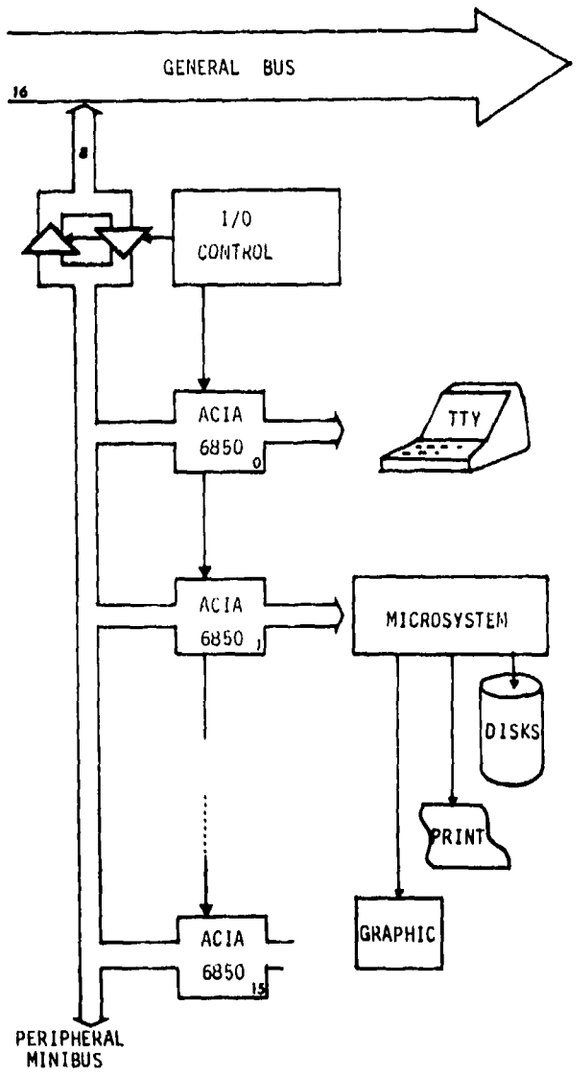


Fig.3 - THE PERIPHERAL MINIBUS

3 - THE MICROCONTROL

3.1. A two-leveled microprogramming

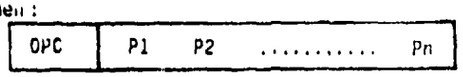
Microprograms, written in LEM, are compiled to produce fixed microcode. Vertical microprogramming used for this implementation results in two advantages : the effort of the compiler is less important and the size of microinstructions can be shortened. This reduces the amount of microcode to swap during control switches.

The great diversity of control signals to provide (in particular, to control the tri-state bus) has led to a two leveled microprogramming. The method used here is different from the nanoprogramming of QM.1[8] which uses a second level of microprogramming. To execute a microinstruction through the datapath one must :

1. provide some parameters :
   - number of A_i , B_i , R_i ...
   - long, short constant
   - number code of branch operation, of ALU function ...
2. define an action to execute, i.e. to state a particular data transfer through the datapath.

The second part, fixed for a given action, still requires much more bits for the direct control of gates. The repetition of such a long "dead-bit" sequence is cumbersome. Thus, the action to be executed is specified by the second level of microprogramming, in a single horizontal word where each control bit drives directly the gates : it is the executive.

The format of a fix-sized microinstruction is then :



OPC represents the code number of an executive, and the P_i's are the arguments.

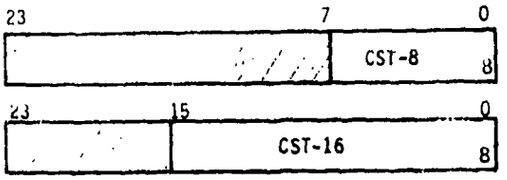
The size of the microinstructions is 32 bits. To the operation code (opc) can correspond up to 256 executives. Theoretically, a great number of executives can be defined but practically the facilities of a datapath are never completely put on use : our simulation of a LISP system1 required 60 executives only. The executives reside in a fast PROM memory (t_A = 35 ns) with 256 words of 116-bit length.

3.2. Description of the microcontrol words

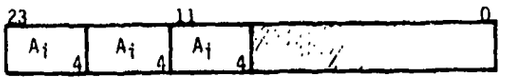
. The microinstruction parameters

There are 10 available p_i parameters. A microinstruction is an assembling of some of these parameters. The assembly rules are stated by each parameter place within the 24-bit parameter field.

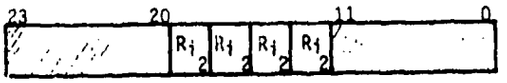
The typical formats are :



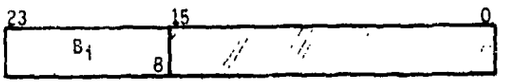
Short and long constants



Three different places are available for the A_i registers

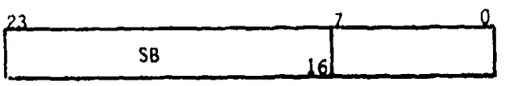


Four different places are available for the R_i registers

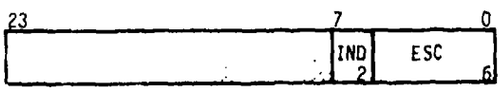


B_i registers

The T_i registers are associated with the BRanch field



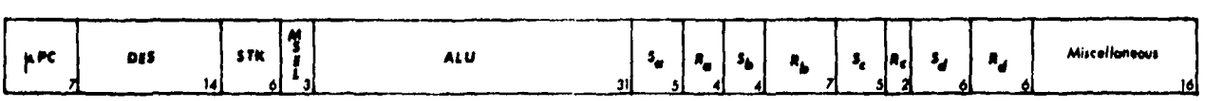
SB specifies the CALLed microprocedure address



ESC is the escape tag and IND specifies the stop mode for the return on escape condition : < , = , >

. The executive word

The executive is divided into 14 sub-fields which can be, or not, attached to a particular control task upon the datapath. The size of the following sub-fields is illustrated below.



EXECUTIVE WORD FORMAT

field name	CONTROL OF
µPC	microprogram counter
DES	MPX (Shift and Mask)
STK	Stack memory
MSEL	Memories selection
ALU	AMD 2903 ALU
$S_{A,B,C,D}$	source selection for the general bus transfers
$R_{A,B,C,D}$	receptor selection for the general bus transfers

(A,B,C,D specify the four different transfer modes, via the general bus)

3.3. Operating

The cycle time of the M3L microinstructions is fixed to 500 ns. It may seem to be long for a modern technology but with regard to the power of microinstructions it is a good speed. The cycle starts with the fetch of the microinstruction (100 ns), it includes some register moves, and always a main control phase which is 200 ns long.

As the case may be, this phase performs :

- an access to the pair-cells memory
- an arithmetical operation on the ALU
- a context switch with an access to the stack memory
- a refresh cycle.

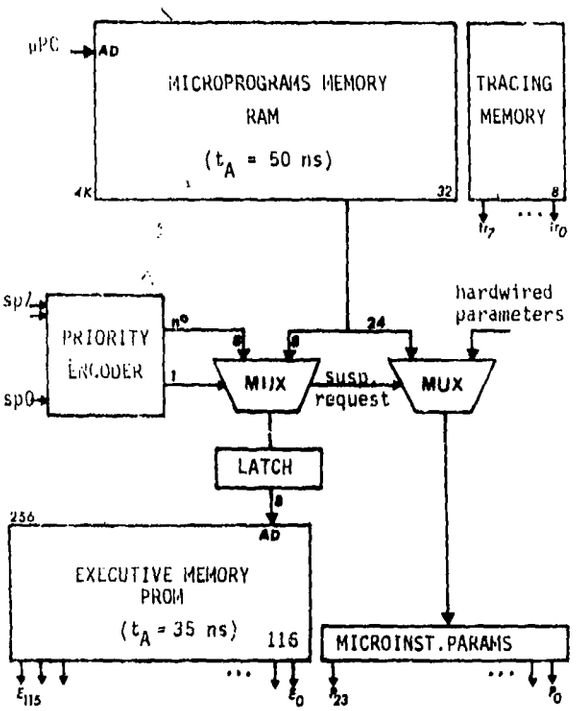


Fig.4 - THE TWO-LEVELED MICROPROGRAMMING

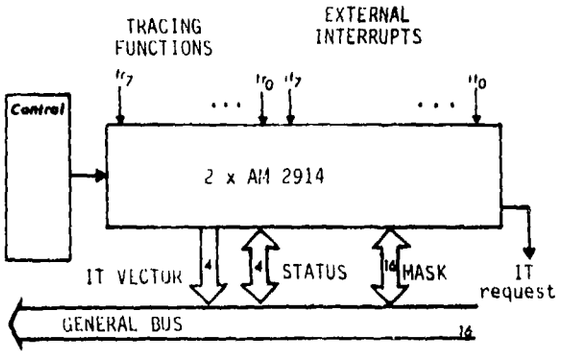
3.4. Suspensions

A suspension is a request for a temporary halt of the current microprogram. During this halt a single microinstruction is performed. The suspension takes place when the latch is loaded : an encoder detects the suspension and yields its number. As there are 8 different suspensions, the 8 first executives will therefore be regarded as suspension handlers.

One of these suspensions will be the refresh request for the dynamic MOS memory. The aim of this suspension is to perform a refresh cycle without modifying the current context.

3.5. Interrupts and microinstruction tracing

Another suspension will be associated with the interrupt request. It has to save the current context without changing the microprogram counter (µPC), also it has to branch to the interrupt handler. At the hardware level, the management of interrupts is achieved with the help of two interrupts controlers (AM 2914) which allow the handling of 16 interrupts levels:



To each microinstruction word, a tracing byte is concatenated, where each bit is associated with a microsoftware interrupt. The bits are setted at the compiling stage. Thus, when running, they activate the corresponding tr_i interrupts which then are held sequentially, according to their priority level.

They can be enabled or disabled in software. They are used in microprograms debugging and for the M3L prototype measurement.

4 - THE PAIR-CELLS MEMORY

The pair-cells memory is the main resource of M3L. It is built with dynamic MOS memory. Each chip contains 16 k - 1 bits and its access time is 150 ns. The pair-cells memory is organized in 40-bit wide words which are divided in three fields having each one 16 , 16 and 8 bits.

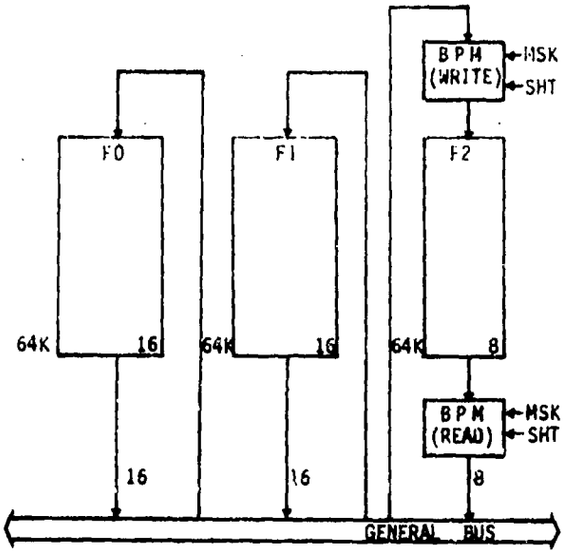


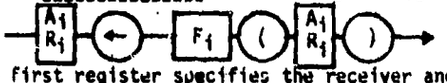
Fig.5 - THE PAIR-CELLS MEMORY

The access to the pair-cells memory, in the read/write mode, is done through the general bus. With respect to data moving there are two kinds of access in the read mode and one in the write mode. As for control there are three kinds of access.

4.1. Access to the pointer field

. Data moving in the read mode

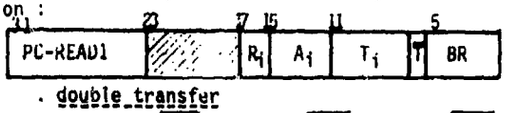
. Single transfer : The LEM syntax is



the first register specifies the receiver and the second one contains the address of the emetter.

Example : A2 ← F1(R3) means :
"read the F1 field of the pair-cell, which address is stated in the R3 register, and store it into the A2 register"

when compiled it yields the following microinstruction :

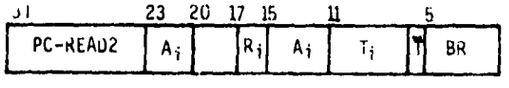


. double transfer

The first register contains the address of the emetter whereas the second and the third registers deal with the receivers of the fields F0 and F1.

fetch A1 into A2 and R3 is equal to $\begin{cases} A2 \leftarrow F0(A1) \\ R3 \leftarrow F1(A1) \end{cases}$

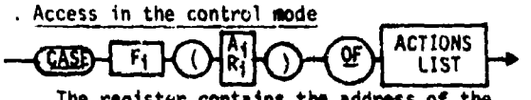
It yields :



. Data moving in the write mode

The first register contains the address of the memory cell to be modified, and the second one contains the information to be moved.

. Access in the control mode



The register contains the address of the referenced to memory word. After reading, the content of the corresponding F_i field is added to the microinstruction address register. In most situations, the access in the control mode concerns the descriptor field of the memory cell. Hence, this multiple branch operation enables the 3L form to be decoded. More details on this microinstruction are given in the section 5.3.

4.2. Access to the descriptor field

Whereas the access to the pointer fields (F0, F1) is fixed, the access to the descriptor field (F2) is more versatile. As a matter of fact, for a given emulated system, this field can arbitrarily be divided into contiguous, or superposed sub-fields. These sub-fields can be accessed to in the read/write, or control mode, like the pointer fields.

Ensuring the access to a sub-field of DES needs a special device to select the field. This device was discussed in a more general situation[9]. Here it is applied to a byte only, thus it is very simple. There is a mechanism for the reading operation, and another mechanism, strictly symetric, for the writting operation. Therefore we will only describe the fetch mechanism.

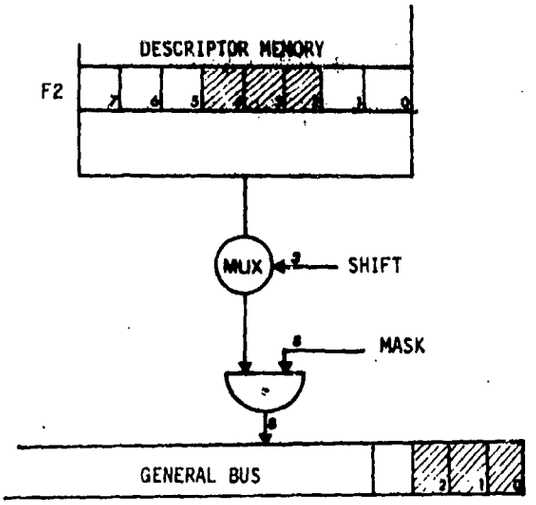
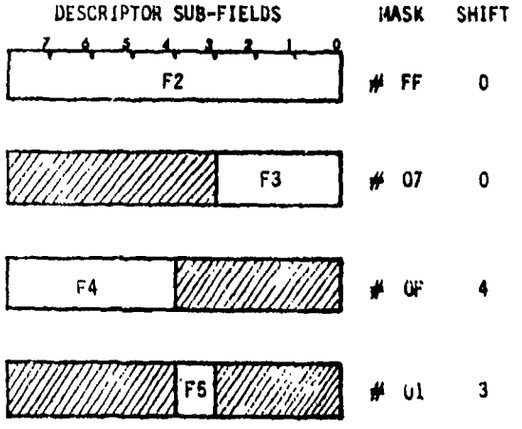


Fig.6 - PRINCIPLE OF THE DES ACCESSING MECHANISM

A first logical level, MUX, performs a circular shift on the descriptor byte. This shift is performed in a purely combinatory an parallel manner by a special chip (SGN 0243). A second logical level masks

the irrelevant part of the descriptor byte. The selection of a field requires the specification of a shift (0-7) and a mask (a byte). These informations are included into the executive of the microinstruction which fetches the sub-field.

Example :

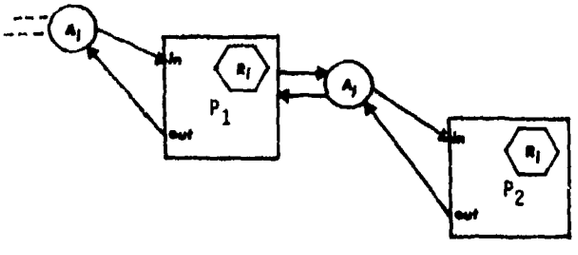


The combinatory nature of the select mechanism of the descriptor sub-fields enables the M3L "memory word" to be viewed as a sequence of fields $F_{i=0,n}$, which are equally accessible in the read, write, or control mode, in a single microinstruction cycle. This emphazises the thorough attention which was paid to the access to the intermediate environment on M3L.

5 - THE CONTROL UNIT

Beyond the special organization of the main memory, the second feature of the M3L architecture concerns its control unit. As a matter of fact, it has to support the recursivity mechanism which is a fundamental aspect of the emulation functions. The LEM language is recursive and this is conveyed through the hardware structure at the level of the control unit of M3L.

A LEM module is composed of little procedures which are independent and not ordered. They can refer to each other and even to themselves. In control switching from a micro,rocedure to another, A_i global registers are used for parameter passing and R_i local registers are automatically saved.



P_1 takes its input arguments into the A_i registers and outputs its results to P_2 via the A_i's. The object of the R_i registers is to maintain the value of A_i registers in the environment of P_1, this value does not have to be erased by the application of P_2.

To the recursivity an automatic escape mechanism is added. Writting the top/down recursive parsers requires such devices wich are similar to software interrupts (like "ON conditions" of PL/1).

An escape microinstruction performs a return operation to the last call microinstruction which has set, in the recursivity stack, a tag number (E_i constant) equal to the tag number of the escape microinstruction. Escapes and recursivity are two concepts which are closely related, hence they have been merged in order to offer a better systematization of the control transfer between microprocedures. It is thus stated that, in LEM, calls are recursive and returns are escapes.

The control unit is illustrated in the fig.7. Its main components are :

ESC stack : enables the escape number to be saved when a recursive call occurs

COMP : when an escape microinstruction is performed, it indicates if the escape number, given as an argument, corresponds to the escape number, that is read into the ESC stack

MPX3 : 32 x 1 multiplexer. The selection is made according to the T_i number, passed as an argument, meanwhile T allows the output to be, or not, inverted

µPC stack : is the saving stack for the microinstruction address register

ADDER : is a simple adder to perform relative branches

MPX1,MPX2 : are the input multiplexers of the microinstruction address register

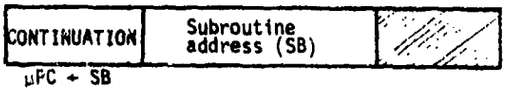
µPC : is the microinstruction address register

µPC control : produces the control signals which correspond to the operation code of the current microinstruction.

The control unit microinstructions

The five basic microinstructions dealing with the sequencing of the microprograms are : the continuation, the conditional branch, the multiple branch, the recursive call and the escape.

1. Continuation



µPC ← SB

Owing to the continuation microinstruction, it is possible to perform branches between the microprocedures without any push operation.

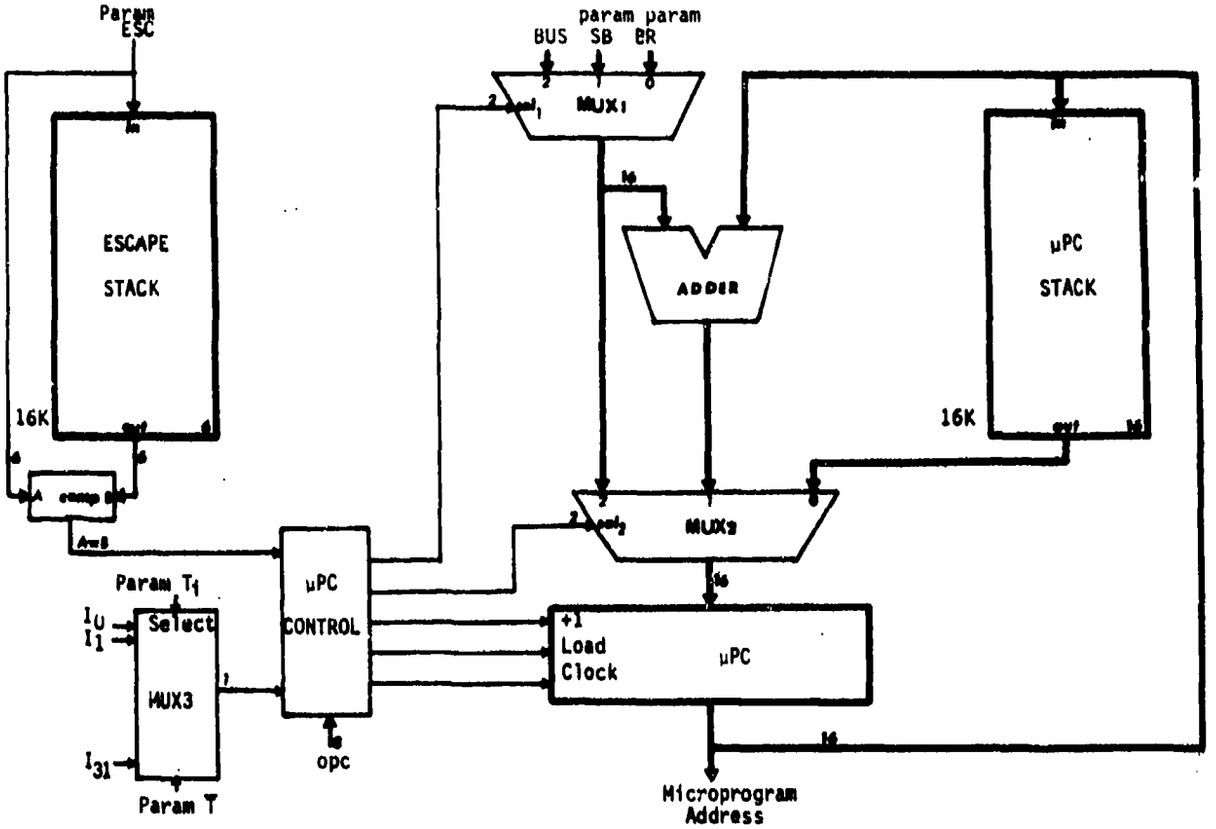


Fig.7 - THE CONTROL UNIT

	T	A = B	+1	LOAD	MPX1	MPX2	STACK
NOP			+1	NO			
CONTINUATION			NO	LOAD	1	2	
BRANCH	T		+1	LOAD	0	1	
	$\bar{T}$			NO			
CALL			+1	LOAD	1	2	WRITE(*)
ESCAPE		=	NO	LOAD		0	READ
		≠		NO			
MULTIPLE BRANCH			+1	LOAD	2	1	

(*) The storage of the μPC into the stack is performed after the incrementation and before the load

Table 1 - THE MICROINSTRUCTIONS OF THE CONTROL UNIT

2. Conditional branch

if ($\overline{T}$ = 1 and T_i = 1) or ($\overline{T}$ = 0 and T_i = 0)

then μPC ← μPC + BR + 1
else μPC ← μPC + 1

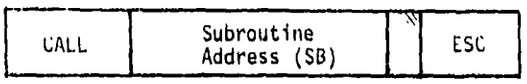
The deplacement BR is signed. The signe bit is in the most significant position.

3. Multiple branch

μPC ← μPC + 1 + $F_i(A_j)$ $\quad (i \geq 2)$

This microinstruction enables a decoding starting from a sub-field (F_i) of the descriptor.

4. Call



V
* μPC ← μPC + 1
* the current context is saved into the stack :
R_i stack ← $R_{i=0,3}$
ESC stack ← ESC
μPC stack ← μPC
* μPC ← SB
V

5. Escape

V
* The context is popped from the stack :
$R_{i=0,3}$ ← R_i stack
* if E_i = ESC stack then μPC ← μPC stack
V

The escape microinstruction is executed as many times as necessary until finding an escape number corresponding to that, specified in the E_i field. It scans the control unit stack in search of its corresponding context. Hence, it generalizes the return mechanism.

CONCLUSIONS

The first remark that we can make about the M3L architecture is related to the numerical processing ; it is not absent, since without it there would not be any execution, but it takes a secondary place. This does not imply that M3L is not able to perform efficiently this kind of processing. On the contrary, owing to the advanced integration capabilities, a LSI family ensures, alone, the functions of the conventional architecture very efficiently.

Whereas the numerical processing can be easily integrated, this is not true for the non-numerical processing. As a matter of fact, it deals mostly with the organization of the information. It does not need any special processor but it is expressed through the distribution of the resources in the computer architecture. On M3L, a special attention was paid to the organization of the resources and in particular to the memories management ; the M3L architecture is based upon two memories : the pair-cells memory and the stack memory.

The M3L project started in september 1977. The prototype, drawn during 1979, is presently in the achievement phase and will be operational in june 1980. The complete machine, with the input/output interfaces for the connecting of the TTY and disks management, is made of five boards following the European standards. The prototype is equipped with a 64 K pair-cells memory and a 16 K stack memory, representing 70 percent of the chips.

The architecture of M3L is simple. Just like the Von Neumann architecture, it varies in direct ratio with the size of the memory. Therefore, it can serve as a basis for a line of general host systems. The present implementation corresponds to a middle need but a new version of M3L, with a virtual pair-cells memory is studied where the datapath will be 24-bit wide. Just like the Von Neumann architecture it offers a systematic approach for the implementation of the direct execution scheme, that makes it easy to understand and to use. Consequently, it bears the required features for a large diffusion. From that time onwards, there is no doubt that such an architecture, and more generally λ-architectures, will supersede the conventional sequential computer systems.

ACKNOWLEDGEMENTS

This work was done at Paul SABATIER University in the Laboratory of Professor R. BEAUFILS and was sponsored by the French IRIA under grant # 79-027 for the building and evaluation of a prototype offering LISP and PASCAL capabilities.

REFERENCES

[1] J.P.SANSONNET, M.CASTAN
Un exemple d'émulateur : M3L
Report LSI # 131 Univ.Paul Sabatier Toulouse
June 1978

[2] J.P.SANSONNET, M.CASTAN, C.PERCEBOIS
Definition et évaluation d'un émulateur
Report IRIA 79.027 Vol.1,2 1979

[3] RICE et al.
Symbol-2R System
SJCC AFIPS 1971

[4] BASKOW, SASSON, FRONFELD
System design of a FORTRAN-machine
IEEE Trans.on Computer Vol.16 n°4 1967

[5] Y.CHU
High level computer architecture
Academic Press 1975

[6] J. Mc CARTHY
LISP 1.5 programmer's manual
MIT Press - Cambridge 1962

[7] J.P.SANSONNET
The 3L-Model : an alternative to the Von Neumann architecture
LSI Report # 77 TOULOUSE January 1980

[8] NANODATA CORPORATION
Computer alters its architecture via new control
Electronics - August 1974

[9] D.LITAIZE et al.
An efficient hardware tool for bit pattern manipulation
EUROMICRO Congress Venice 1976

[10] M.L.GRISS, M.R.SWANSON
A microprogrammed LISP-machine for the Burroughs B1726
SIGMICRO NEWSLETTER Vol.8 n°3 1977

[11] A.BAWDEN, R.GREENBLATT, J.HOLOWAY
LISP-machine progress report
MIT Report # 444 August 1977

[12] L.W.HOEVEL
"Ideal" directly executable languages - An analytical argument for emulation
IEEE Trans.on Computers Vol.C-23 n°8 1974

[13] E.I.ORGANICK, J.A.HINDS
Interpreting machines. Programming of the B1700-B1800 serie
The Computer Science Library, North Holland, 1978

HIGH LEVEL ARCHITECTURE FOR A REAL TIME LANGUAGE LTR

G. DURRIEU*
B. FROMENT

F. CUAZITI.**
B. LECUSSAN
J. ROMAIN

D. VIDAL
P. VUARIER

*CERT/DERI 2 Avenue Edouard Belin
31055 TOULOUSE CEDEX
FRANCE

**UPS/LSI 118, route de Narbonne
31077 TOULOUSE CEDEX
FRANCE

ABSTRACT

This paper presents a methodology of definition of a high level machine for a real time language. First, the choice of an indirect execution computer architecture for this class of language is discussed.

Apart from the algorithmic aspect already examined in previous realizations, this type of language creates problems of management in a multi-task environment, of definition of the concept of interruption on a high level machine and of implanting complex systems which require a structured conception.

An application of the defined methodology is described which consists of the definition and realization of a high level machine for the LTR language, insisting on the implementation of problems specifically linked to real time.

INTRODUCTION

The design of a general-purpose computer usually precedes the design of the software tools it is intended to support ; software and hardware interfacing is performed by instructions in the machine language managing the physical resources of the computer. The implementation of a high level language on a general purpose computer calls for, therefore, the presence of translators which produce (compilers) or use (interpreters) these instructions.

The semantic gap between the external form of a high level language and the machine language infers very complex, expensive translators which are not necessarily free from errors.

In the last 20 years many high level languages adapted to programmer's needs have appeared which have been implemented with the help of compilers.

At present a large number of high level languages exists which correspond to most programming needs.

The definition of a data processing system (computer + language) may, therefore, move in a new direction : given a chosen programming language, let us define a computer architecture associated with this language.

This approach is attractive for two fundamental reasons :

- choice of the language which best expresses the problems to be dealt with (FORTRAN for scientific calculations, COBOL for management decisions, PASCAL for general applications, ...)
- the efficiency of an architecture designed specifically to support the language.

In recent years, many studies have been carried out based on languages which are, essentially, algorithmic (FORTRAN, PASCAL, EULER, BASIC, SYMBOL,...). The study described in this paper concerns the definition of an architecture specialized in the execution of a real-time system. The fact that a real time application is taken into account introduces some specific problems :

- the programming system is composed of very numerous (> 500) interacting programs ; therefore, on the one hand, there is an extremely large volume of source programs (in the region of several hundreds of thousands of instructions) and, on the other, the problems of synchronization between the different tasks are crucial
- task switching must be efficient so that an internal or external event can be enable as quickly as possible
- the computer must allow separate execution of the different tasks so as to ensure a structuration of the application.

I - INDIRECT EXECUTION ARCHITECTURE

Indirect execution architecture, is made up of two distinct parts :

- a software module, which produces an intermediate language based on the source language
- a hardware module, which execute this intermediate language.

The crucial point of this approach is the definition of the intermediate language (IML) which must be sufficiently close to the source language if the compiler is to remain simple, and sufficiently close to the hardware if the execution must be efficient.

It follows, therefore, that there cannot be an general purpose IML adapted to every machine language and architecture. The definition of such an architecture must, therefore, start with the definition of this intermediate level.

Separate module compiling thus demands existence of a linkage editor to generate an executable system.

Two solutions may be envisaged :

- the edition of static links takes up the concepts which exist on conventional machines and furnishes an executable module
- the edition of dynamic links is carried out at the execution time ; in this case, when the resident system meets an external reference, it must enter the module in central memory and start the execution. This procedure, which includes an address computation, is time costly.

The choice between these two techniques depends on the source language organization and the constraints of execution time.

Direct execution computer architecture, on the other hand, can support the execution of a high level language without any change of the original text. This approach presents many advantages (suppression of all the software system, the compiler, the linkage editor, the loader ; interactive program debugging[5,6,7]) for a certain type of application ; this layout seems to be difficult to implement for complex systems, notably for multi-task real time systems. For example the definition of interruptible points in such a layout is rather delicate : an interruption can be enable either at fixed points in the execution of a source instruction (at the beginning or at the end), and, in this case, the masking time may become too long to comply with the system specifications, or at each analysed token and, in this case, the processor context may become too voluminous and context switching inefficient.

2 - HIGH LEVEL ARCHITECTURE FOR A REAL TIME LANGUAGE

The need for efficient execution, the management of a multi-task environment and the complexity of the real time systems involved lead to the choice of an indirect execution architecture to support the execution of these systems.

This methodology, essentially interpretive,combines the advantages of the compiling and interpretation techniques.

The source text is translated into a coded text, compact and syntactically correct, whose execution may be restarted, postponed or linked with other modules.

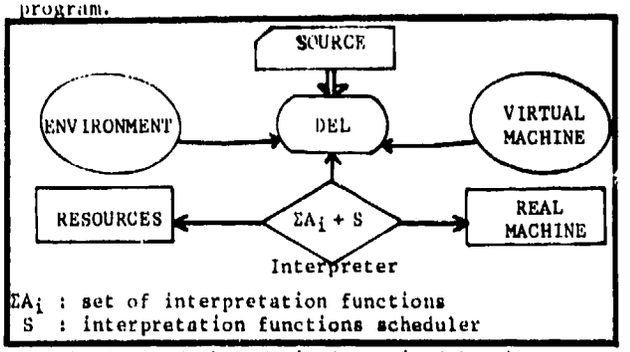
The intermediate text is interpreted with the help of microprogramming techniques on a data path adapted to its interpretation.

This methodology avoids the two basic reproaches which are levelled at compilation and interpretation. The compiling phase is simple, since it does not realize code generation and optimization as in classic compilers. Moreover, the text produced is independent of machine resources (memory, registers,...) and the semantics of the instructions are close to the source language.

The interpretation of such a language level may be efficient thanks to microprogramming. Classic programmed interpreters were not very efficient as they were in the central memory and they acted on rudimentary data paths (adders, registers).On the other hand, a microprogrammed interpreter is in control store (with an access time about 10 times faster) and present day technology allows the creation of data paths better adapted to interpretation.

2.1. Intermediate machine language

The compilation phase must make the source text directly interpretable. The properties of these DEL (Directly Executable Language) have been largely defined by L.W. HOEVEL[9]. This phase comprises, therefore, a syntactic and semantic analysis of the source text, symbol processing, processing of forward references and labels and the prefixing (or postfixing) of the instructions. This processing may be defined as a transfer from a concrete machine (source text), defined by a concrete grammar, to an abstract machine (DEL), defined by an abstract grammar, used by the interpreter to execute the abstract program.



ΣAi : set of interpretation functions
S : interpretation functions scheduler

The form of the IML is determined by the nature of the language ; however, some characteristics may be singled out. The transfer of the source program into the virtual machine brings about an environmental change. An intermediate environment may be composed of three types of space :

- program space
- descriptor space
- data space

The program written in IML is a finite series of binary fields, of varying length. These fields are the operation codes, operand identifiers, descriptor space references or constants.

The descriptor space contains all the semantic information on the data, and, notably, the type and the access mode to the data space.

The data consist of information of varying length. They represent arithmetic values, texts, system information (events, semaphores) or procedural parameters.

2.2. Characterization of interpretation processing

Interpretation processing comprises three types of processing :

- organic processing associated with the management of the tasks making up the system (activation-deactivation) and managing the machine resources
- formal processing associated with an execution control managing the execution of a task
- effective processing associated with the final execution of the instructions.

The central processing unit of present-day computers are defined solely to the execution of effective processing.

A high level architecture must, therefore, be made up of hardware structures in order to support efficiently formal processing and organic processing. These structures must permit a description of program algorithms at a macroscopic level ; that is, at the level of the algorithmic logic.

Effective processing, on the other hand, permits a description of the algorithms at a microscopic level ; that is, at the level of functions realization.

APPLICATION

HIGH LEVEL ARCHITECTURE FOR THE LTR LANGUAGE*

LTR is a ten years old real time language whose application are now implemented on classic computers (MITRA, IRIS,...) through the intermediary of a compiler which produces a symbolic text which must be assembled on the target machine.

This implementational outline is not very efficient at the compiler level nor at the code generation.

On the other hand, this language is complete enough to be able to express most of the problems of a real time application. Therefore it has been chosen by several departments of the French Defense Department for writing real time systems.

The problem is the definition of a machine architecture which can support its execution efficiently. We shall, therefore, examine an indirect execution computer architecture to execute LTR even though this is a compiler oriented language.

1. PRESENTATION OF THE LANGUAGE

LTR, Real Time Language (Langage Temps Réel) is a high level programming language destined for systems realization. It presents a highly structured organization shown by a partition into ARTICLES at the highest level. A LTR system is a set of ARTICLES.

1.1. Types of articles

Data articles are of three types :

- DATA ARTICLE : data shared by a program and its subroutines
- GLOBAL DATA ARTICLE : data common to the system data set
- SYSTEM DATA ARTICLE : data specific to the system environment.

The processing articles describe the algorithms concerning the data declared in the data articles or in the processing articles.

There are three types of processing articles :

- PROCEDURE ARTICLE : corresponds to the concepts of subroutines or functions
- PROCESS ARTICLE ; describes a process running in a multi-task context (concept of software task)
- INTERRUPT PROCEDURE ARTICLE : describes a process, whose execution is tied to the interruption system (concept of an immediate task).

1.2. Structure of a LTR system

Figure 1 describes a LTR system ; the separate compilation of a task may be carried out, the compilation unit being :

<SYSTEL DATA ARTICLE><GLOBAL DATA ARTICLE>*<EXTERNAL GLOBAL PROCEDURE>*<EXTERNAL PROCESS>*<task>

Program procedures may be called only by those of the same task.

A task may activate another task and take back control at the end of execution (closed call) or lose this control to the advantage of a task with higher priority (open call).

* This work is supported by the Direction des Recherches et Etudes Techniques (DRET) of the French Defense Department, at the department of Computer Science of the Paul Sabatier University and the Department of Computer Engineering (ONERA-CERT) of the Centre d'Etudes et de Recherches de Toulouse.

The implemented system must ensure local procedure recursivity and task reentry.

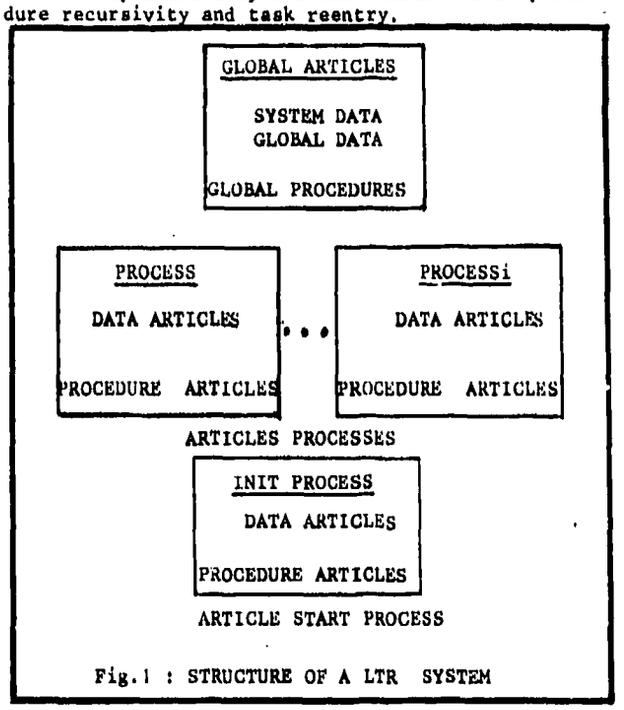


Fig.1 : STRUCTURE OF A LTR SYSTEM

The range of the identifiers outside the processing article is as follows :

. the only accessible data are those declared in :
  - the task DATA ARTICLES
  - GLOBAL DATA ARTICLES
  - the parameters
. the only usable ones are :
  - the task PROCEDURE ARTICLES
  - the GLOBAL PROCEDURE ARTICLES

Inside the article, the classic block structure rules must be respected.

1.3. Principle of data allocation

In LTR, lead to different data storage allocation the type of article and the data organization.

A. Static and permanent

These are the data, tables or structures declared in a GLOBAL DATA ARTICLE or in a DATA ARTICLE. The store space is reserved by the compiler and life expectation is linked with that of the task.

B. Automatic allocation

These are the data, tables or structures locally declared in the processing articles. The data are dynamically initialized and data overlay takes place according to the block structure. Life expectation is linked to the internal block in which they are declared.

C. Controlled allocation

This concerns virtual data pointed by the user. The data are described in a data or processing article : links between the description and the data zone to which they apply is realized by the execution of pointer manipulation instructions or by storage allocation.

D. Chain allocation

This concerns sets pointed by the user but whose chaining is automatically ensured by the allocation

CODE	Param 1	Param 2	Param 3 and others	FONCTION	Notes
AFF	operand	opde or constant		Affectation	
ADD	operand	opde or constant	opde or constant	Param 1 = Param 2 + Param 3	(1)
LSS	operand	operand		Comparison of Params 2 + 3 and affectation of result (booleen) to Param 1	(1)
IF	address 1	address 2	address 3	(f 1)	(2)
FOR	address 1	address 2		(f 2)	(2)
WHILE	address 1	address 2		(f 3)	(2)
CALL	operand		(opde or constant)*	Param 1 : descriptive of procedure Param 3 : parameter list	
CALLP	operand	entry TD		Entry TD : address of a TASK DESCRIPTOR Params 2+3 : identical	
NEW	operand	operand	opde or constant	Insertion of an element in a set Param 1 : set 2 : insertion address 3 : name of element to be inserted	

(f 1) IF <a_1><a_2><a_3> <exp.bool.block> <THEN block> <ELSE block>

(f 2) FOR <a_1><a_2> <incr.block + test> <FOR block>

(f 3) WHILE <a_1><a_2> <exp.bool.block> <WHILE block>

(1) Parameter 1 may be an intermediate variable produced by the compiler
(2) The addresses are N-uple addresses

mechanism. The data are described in a GLOBAL DATA ARTICLE.

This presentation of the language fixes the constraints on defining memory management for a LTR machine. We shall present the solution chosen for implementing such a system below.

2. INTERMEDIATE LANGUAGE (DEL) FROM LTR

An intermediate instruction is a byte chain of varying length called N-uple.

A N-uple may be an expression (OPERATOR,(OPERAND)*) in which the number of operands is fixed only by the LTR instruction specifications.

Definition of the operator codes is fixed by the LTR instructions ; each instruction has been regrouped in the form of an N-uple, at the same time conserving all the semantic contained in the source instruction.

The upper table gives some examples of N-uples.

In the operand part, we may find either a constant, an N-uple address, or a data descriptor address. The operand is prefixed by a directive which prescribes the descriptor type :

- (ch) 'corchain' for bit chains
- (ix) table index
- (rf) reference of a structure field
- (pt) pointer to a set
- (ct) constant
- (op) operand
- (cv) conversion

The DEL-LTR may be summarized schematically as follows :

```
LMI ::= (N-uples)*
N-uple ::= (OPCODE,(OPERAND)*)
OPERAND ::= (CTSI)(OPDE),(CONV/MOD,(CONV))
OPDE ::= op,H.S.H.,(INDEX)
CONV ::= cv NUMBER
MOD ::= ct,(CTSI/OPDE,CTSI/OPDE)/pt,H.S.H.,CONV
INDEX ::= ix,H.S.H./indexi,CTEi
CTSI ::= ct,CTEi
H.S.H. ::= address of descriptor
CTEi ::= immediate constant
```

This intermediate form is very close to the source language. The semantic information contained in an LTR instruction has been coded in the intermediate instruction so as to facilitate interpretation : the interpreter will analyse instruction prefixing by operational code, execution and control addresses and operand directives.

All non-constant variables are addressed through a descriptor which contains the information set characterizing the data used by the interpreter.

The basic descriptor is a 10 bytesword which may have extensions for complex operands (table, structure, process descriptions). In the standardized part, it contains :

NAME	INDIC	BASE	DEPL.	TYPE	STRUCT	SIZE	SCALE	EXT

NAME : reference to a file containing the symbolic name of the variable,this information allows the editing of the state of the variables during, the debugging phase
INDIC : data implantation type : global,local,parameters
BASE-DEPLACEMENT : data implantation address
TYPE : Integer, Real, Fixed, Index, Character string, logic, boolean, quality, static reference,virtual data reference, set element reference
STRUCT : array,structure,structure array,virtual data, set
SIZE : space occupied by the data
SCALE : normalization factor
EXT : pointer to an extension descriptor

3. LTR PROCESSOR STRUCTURE

The LTR processor structure follows from the methodology described above.

The processor is composed of two pipe-line units, one for macro-interpretation processing (MAI), the second for micro-interpretation processing (MII) (fig.2).

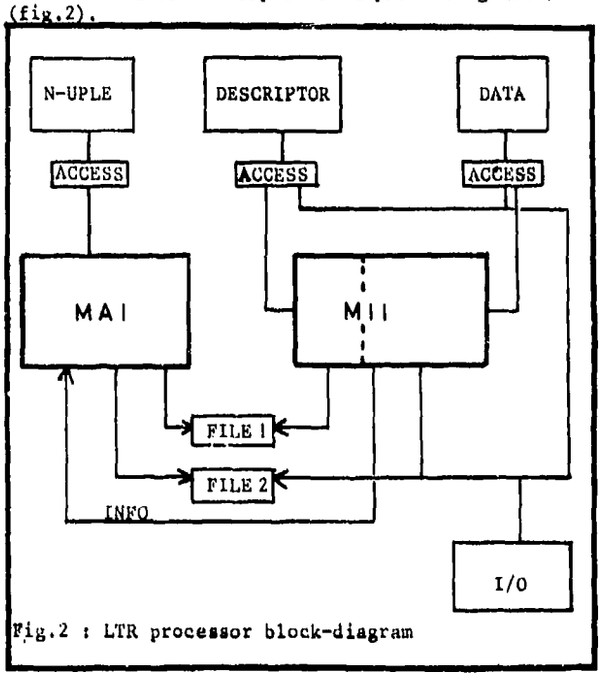


Fig.2 : LTR processor block-diagram

The central memory is divided into three physically separate memories :

- the N-uple memory contains the intermediate code and is accessible to the MAI processor only
- the descriptor memory contains the data descriptors, systems data and processes : it is accessible to the MII processor only
- the data memory contains the data described in the source program.

A N-uples is interpreted in two phases :

- the first, in the macro-interpreter (MAI), manages the IML execution control ; it divides a N-uple into simple instructions which it sends to the micro-interpreter (MII)
- the second phase, therefore, takes place in the micro-interpreter (MII) which merely executes, sequentially, the actions send by the MAI : search for operand descriptor, conversion of a number, arithmetic operations ... ; these actions correspond to a set of microprograms contained in the MII control store.

The connection between the two units is realized through the intermediary of two hardware queues: a parameter queue and a action number queue. Moreover, state variables and calculation results may transit between the two units.

The two queues allow a synchronization of the two processors and ensure pipe-line management.

The division of the stores in function of the information they contain allows a real parallelism between the different accesses and also particularisation of each access :

- the N-uples memory has read access over 4 bytes ; the descriptor memory has a double read/write access also over 10 bytes ; the first contains the descriptor and the second the context of the micro-machine
- the data memory has a read/write access over two bytes, the size of the data path being 16 bits.

The scheduling algorithm occurs on the Micro Interpreter which sends a task number to the MAI ; the context set is described in the CONTEXT section.

3.1. Macro Interpreter Structure (fig.3)

The macro-interpreter supports the formal and organic processings attached to the system execution control. Formal processing amounts to management of the N-uple ordinal counter (management of the recursivity of IML instruction) and organic processing concerns procedure context switching. A context switching may occur on two types of event :

- switching on interruption
- switching on process call

In the first case, the interrupted process contexts may be managed in stacks ; interruption mechanism can be implemented according to a hierarchic algorithm.

When the process attached to the interruption of level takes place, it can be interrupted only by an interruption of level j ($j > i$) ; control will be returned, after processing of level j, to the level i process or to a process with a higher priority.

This mechanism may be implanted with the help of just one stack, the summit context being the active context.

On the other hand, for process activated by an open call, it is possible to avoid returning to the calling process. A stack must, therefore, be allocated to this process and, during switching, the number of the stack containing the caller's context must be saved . The task is, then, executed in its own stack space. For all closed calls, the context may be safeguarded in the active stack (mechanism identical to that of activations on interruption) and for pseudo-open calls (an open which return control to the calling process) two stack spaces are sufficient.

We allow for 16 stack spaces (15 + interruption) which permit an interleaving of 15 open calls without return to the calling process. The size of each space is assessed at 1 Kwords. This space and the management mechanism are represented by stack II. The micro-interpreter context will be switched at the top of the active stack, the active stack being found in the process descriptor.

Ordinal counter management is ensured by a reentrant microprogrammed interpreter whose essential functions are :

- access to the source text
- analysis of the instruction operation code
- to break up an N-uple into elementary ACTION functions.

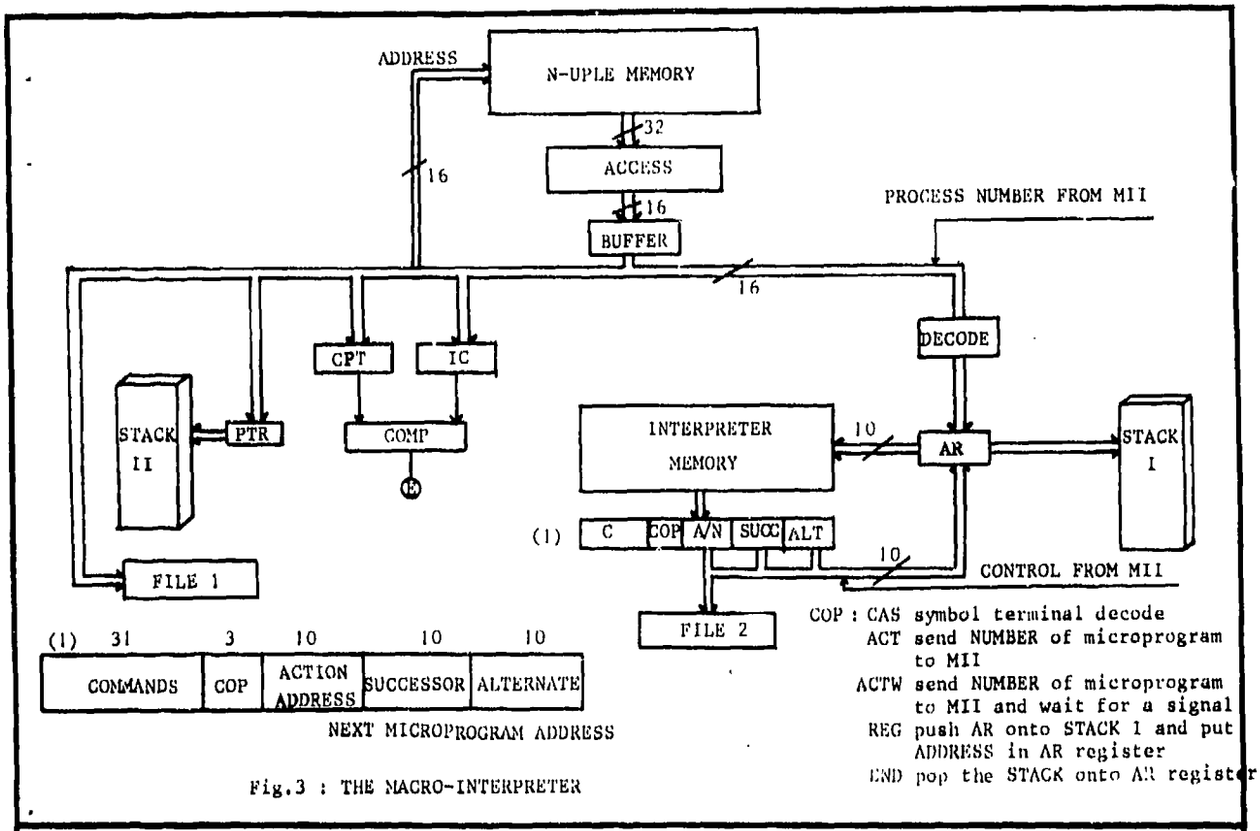


Fig.3 : THE MACRO-INTERPRETER

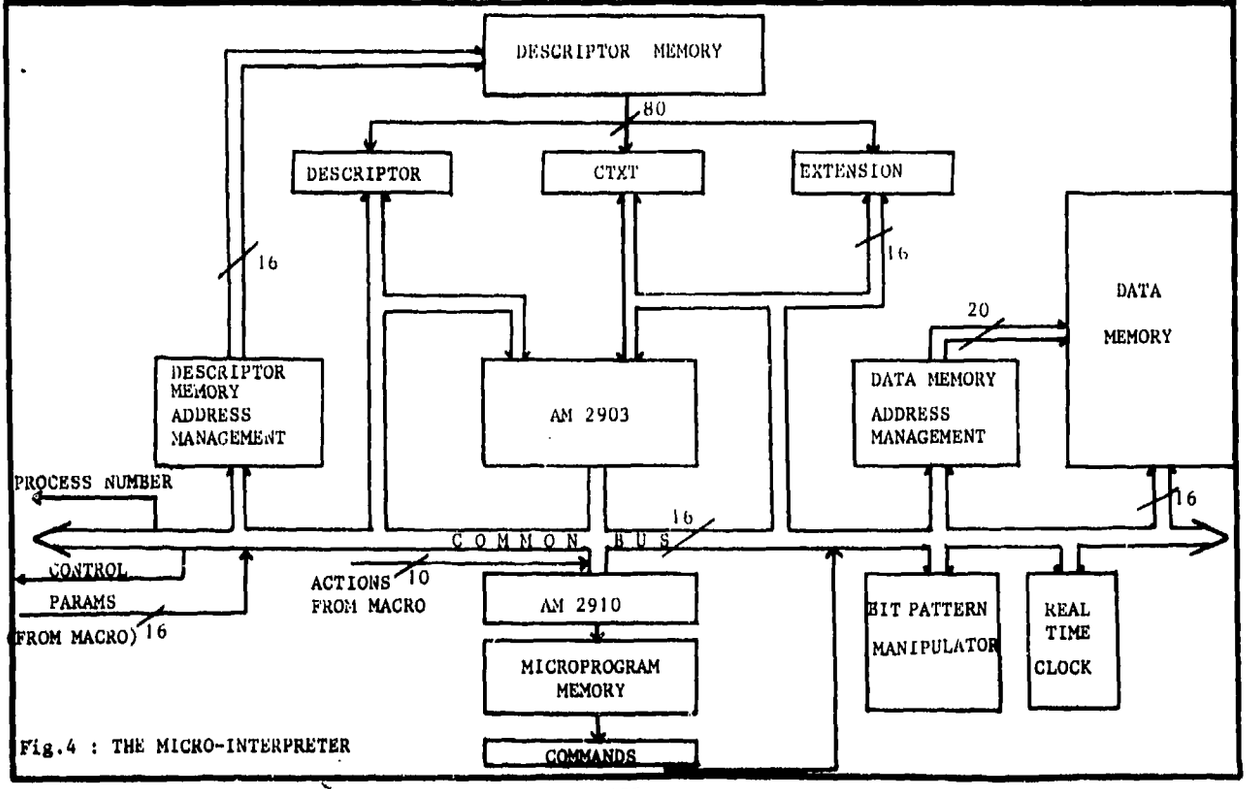


Fig.4 : THE MICRO-INTERPRETER

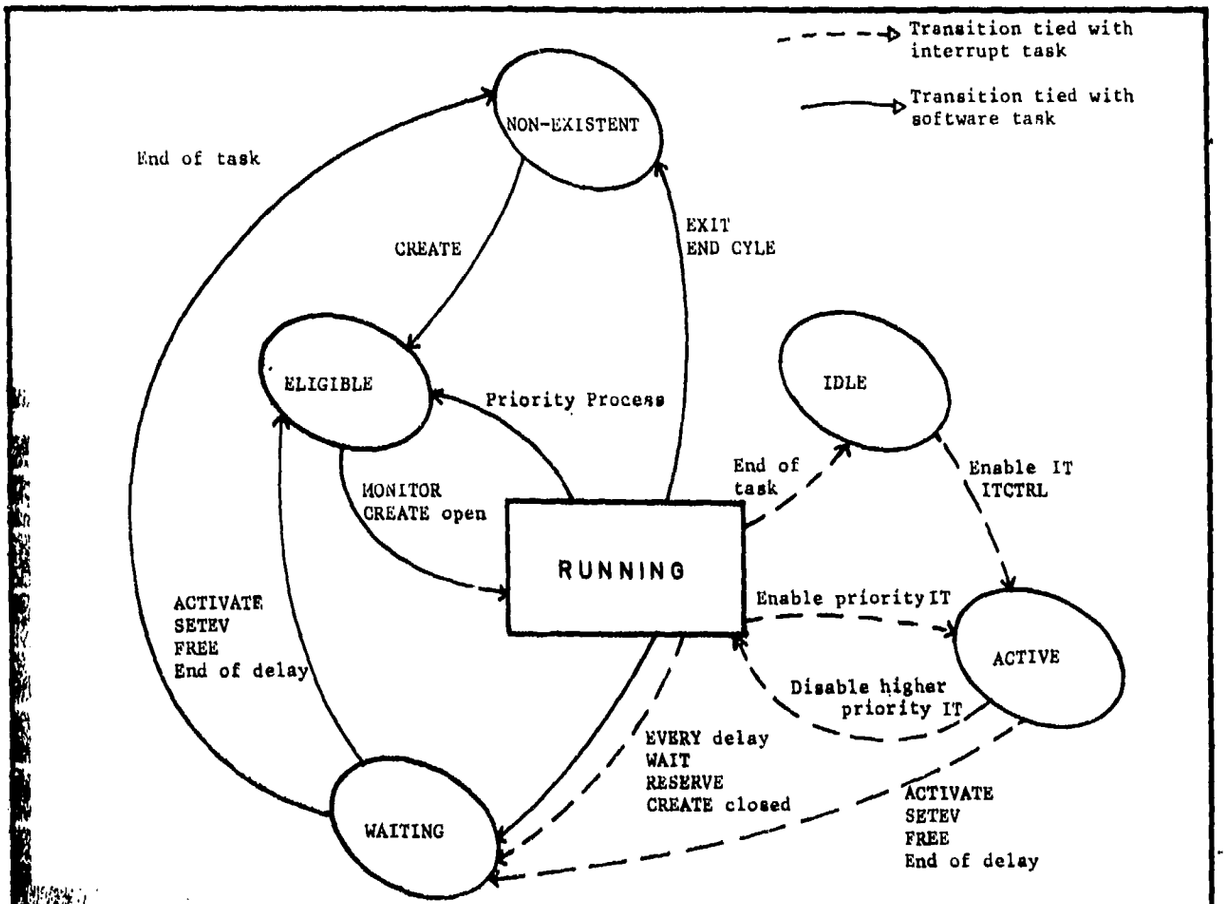


Fig.5 : THE SCHEDULING PRINCIPLE

Example : Interpretation of an IF instruction.

When the operation code is decoded, the interpretation consists in :

- stacking the three addresses $<a_1><a_2><a_3>$ in stack II of the active procedure
- loading $<a_n>$ onto the CPT register
- calling a rule <boolean expression> . (1)

The end of the <Expbool block> is supplied by the comparator which determines the egality between the CPT register and the instruction counter (IC).

Depending on the value of the boolean transmitted by the MII, address a_2 is loaded onto the IC and address a_1 is loaded onto the CPT (value 0) or address a_2 is loaded onto the CPT and the IC register is not affected (value 1). At the end of <block THEN> , address a_3 is loaded onto IC.

(1) This call is carried out by stacking AR onto Stack I and the return of the rule provokes a pop operation. This mechanism allows an interpretation of the language in accordance with a method of descending analysis.

3.2. Micro-Interpreter Structure (fig.4)

The micro-interpreter is the CPU of conventional computers. It is composed of a control store containing the set of interpretation microprograms and a data path formed by an arithmetic logic unit (AMD 2903) and a Bit Pattern Manipulator (BPM) capable of performing logic operations on bit sets (permutation of bytes, extraction and scaling of bit fields, concatenation). The MII manages access to descriptor and data stores and executes the part of organic processing relative to the management of the data space of a procedure.

The access register of the descriptor store is, in fact, a local memory composed of three blocks of ten bytes. This memory constitutes an extension of the internal registers to microprocessor AMD 2903.

The first block contains a procedure descriptor or a data descriptor, the second may contain a data descriptor, and the third contains the MII context. We shall see in the CONTEXT section that this solution allows an optimisation of context switching.

4. STORE MANAGEMENT

4.1. Data store management

Logically, this store should be managed in such a way that the implantation of data and way of acceding to it should be directly deducible from the LTR system structure and from the constraints quoted in (1).

The structuration of the program into ARTICLES suggests an addressing in relation to different bases. This technique allows, moreover, the definition of a protection for each segment, an important factor in the real-time field.

It will, however, be necessary to allow for direct addressing in particular for the passage of parameters by address.

Since the LTR processor takes the recursion and reentry of the procedures and processes into account, it leads us to allocate a stack for each process where the contexts of each procedure call will be conserved and local data of the called procedure will be created.

It can be seen that the basic addressing is not sufficient to manage the memory efficiently. There is a possibility of a proliferation of zones of dynamically created data. It follows that it will be difficult to recover the free space and for this reason we have added to the addressing system a system of storage allocation by paging and "topographic" store.

However we have also tried to adapt the addressing mode to the type of accessed data by addressing directly the global data, whose life expectancy is that of the system, and reserving topographic addressing for data with a shorter life. The characteristics of these different zones are determined by the requirements of the LTR system to be executed.

To sum up, we have allowed for the following addressing modes, which appear in the descriptions of the system variables :

- general direct addressing, for the use of data declared in GLOBAL DATA ARTICLE
- direct addressing for the use of the process or procedure call parameters and also the sets
- topographic addressing, localized in the process, for the use of data declared in DATA ARTICLE
- topographic addressing, localized in the procedure, which interests the process stack, for the use of data declared in a PROCEDURE ARTICLE, global or not.

Different address calculations

Let f_{TOPO} be the function calculating the real address of a variable from its virtual address. This association function consists in replacing the virtual page number by the real page number. This association is realized during storage allocation, by the operating system and is materialized by a "topographic" store. The list of pages allocated to a process is part of its context.

Therefore :

- Calculation of a general direct address (GDA)
  a = (base G) + deplacement
- Calculation of a reference direct address (DA)
  a = deplacement

An address of this type is always contained in a pointer :

- Calculation of a process local address (PSA)
  $a = f_{TOPO}$ ((base L) + deplacement)
- Calculation of a procedure local address (PDA)
  $a = f_{TOPO}$ ((base Z) + deplacement)

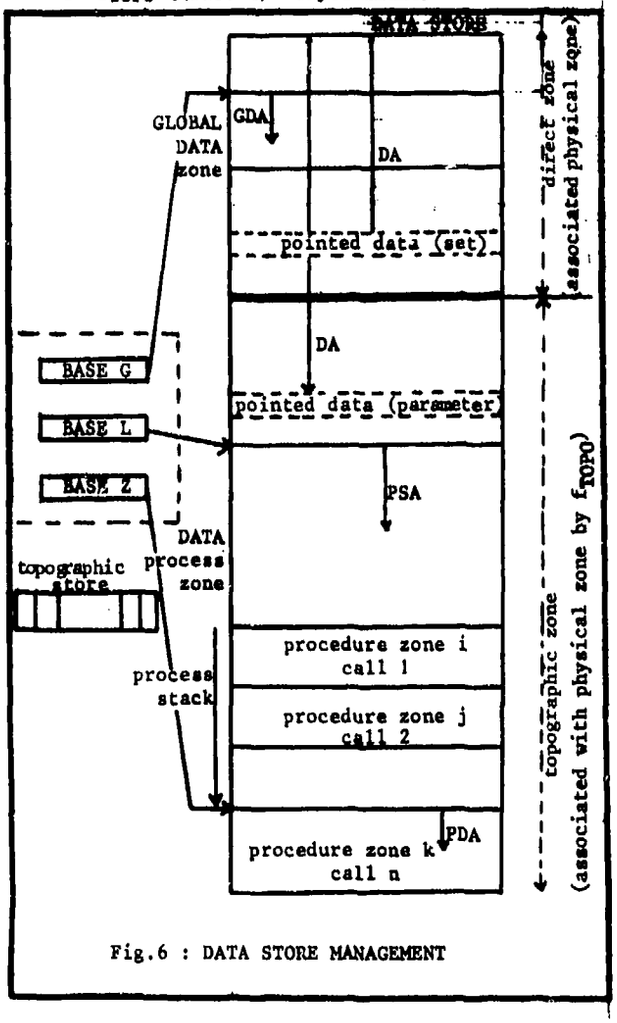


Fig.6 : DATA STORE MANAGEMENT

4.2. Descriptor addressing

The data of a program are referenced in the code through the intermediary of a descriptor. It is implanted in a memory 10 bytes wide and addressable on 64 K. However, in order to simplify program debugging, the LTR source text may be compiled by modules (an executable system may be composed of several modules). The solution classically adopted in machine languages to assemble the different modules consists in making the process linking dynamically. We have not retained this solution as it has proved to be too time costly in execution and considerably increases the system overhead time. We have, therefore, chosen, to address the descriptors by (base, deplacement). Therefore, at a given moment we have three bases :

- Base of Global data descriptors
- Base of Data descriptors
- Base of local data descriptors for active procedure.

The values of these bases are determined when loading the blocks they reference. It is to be noted that these bases are an integral part of the process context.

4.3. Implementation of data systems

We shall now examine the solutions adopted for the implementation of the system processors, notably: the scheduler, management of events and semaphores and interrupts.

4.3.1. Processor implantation

The processors monitors are microprogrammed and run on the micromachine. The data manipulated by these programs are implanted in the form of descriptors, for protection purposes. In effect, only the microprograms are authorized to write in the descriptor store during the execution of a system. These processors manipulate descriptor strings.

4.3.2. Implantation of scheduler data

The scheduler manipulates process descriptors. These descriptors have the following structure.

NAME	INDICS	LAV	LAR	EXT BASE	CODE BASE	DESC DATA
BASE	PROG	SPACE	PROINIT	STACK NUMBER		

NAME : pointer towards process identification
INDIC : process current state word
LAV-LAR : stringing of process in queues
EXT : pointer towards an extension
BASE CODE : address of code implantation
BASE DESC DATA : address of data descriptors
BASE PROG SPACE : address of data
PROINIT : pointer towards procedure status descriptor

The scheduler manipulates only the CU processor's queue (ready processes). In effect, the other lists are manipulated by the other system processors which will return control to the scheduler at the end of their execution. The head of this list is represented by a descriptor implanted in a fixed address with the form :

FIRST LAST	NB LIST	NB CREATED	NB ACTIVE

FIRST,LAST : reference points on the list
NB LIST : number of processes in the list
NB CREATED : number of processes created
NB ACTIVE : number of active processes at present.

4.3.3. Event and semaphore management

We first decided not to implant event expression resolution. Our choice was motivated by the complexity of such a resolution and the multiplication of hardware it would cause. We have, therefore, grouped the processing of events and semaphores. The physionomy of the descriptors manipulated is as follows :

NAME	VALUE	TYPE	FIRST	LAST

NAME : pointer towards the semaphore or event identifier
VALUE : value of an instant of the variable
TYPE : event/semaphore
FIRST,LAST : processor queue reference

4.3.4. Interruption management

The interruptions are materialized by a descriptor with the form :

INTERUPT	STATE	ATTACH PROCESS ADDRESS

The IT descriptors are implanted in addresses equal to running level (IT N° i → descriptor of address i). When an IT is enable, the IT processor inserts the process at the head of the queue. The scheduler takes control and, if necessary, activates.

This processing concerns IT directly connected with a task.

5. INTERRUPTIBILITY

The definition of interruptibility at a "logic" level, that is, at the level of the intermediate language and the macro-interpreter, is very delicate, or, even, impossible, given the contextual interpretation mode we have chosen. An "instruction", or execution unit, at this level, is, in effect, something of variable length, and may even be the program itself.

The concept of point of interruptibility must, therefore, be more closely defined, even if the macro-interpreter level presents the interest of reducing context volume to a minimum when enable the interrupt.

The division of an N-uple by the Macro-Interpreter into ACTIONS permits the interruptible points to be fixed at the beginning of each ACTION. This choice establishes a compromise between the volume of information to be saved and the time needed to set up this safeguard. In effect :

- The fastest possible takeover of the interrupts will have for effect the switching of a larger number of data, therefore an effective time such that this politic is in danger of losing its interest
- A takeover defered until certain key moments in the execution of a program will entail the manipulation of a smaller amount of data and may, therefore, be more efficient than immediate processing.

Moreover, at the beginning of ACTION, MAI context is at a minimum. However, to justifie this choice, the execution time of an action must remain compatible with the requirements of interrupt processing.

6. CONTEXT

Given the machine structure we have described, this context will be larger than that found on a conventional machine. It is, moreover, spread over several functional units and, thus, may be divided into three parts :

- task characterisation context
- macro-interpreter context
- micro-interpreter context

6.1. Task characterisation

This is the part of the context which is closest to the information found on a classic machine. It defines, both the identity of the process and its work space for anything concerning the data manipulated.

Definition of process identity includes the following information :

NAME : pointer to the name of the process
ADCODE : process start address
NIT : tied number of interrupt

This information will be contained in a specific location in the descriptor memory.

The definition of process work space includes the following informations:

ADDESC : description space base
STACK : number of the execution stacks in the Macro-Interpreter
BASE 1 (G) : process global data base
BASE 2 (L) : process local data base
BASE 3 (Z) : local data base for running procedure
BASE 4 : address of page table for the process.

The type of topographic implantation chosen (see above) calls for the constitution of correspondance tables, virtual pages → real pages, proper to each process. During execution of a process this table is loaded in a specialized memory and must exist in memory so that it can be reestablished after interruption followed by context switching.

6.2. Macro-Interpreter context

The execution of a process brings about an evolution of the information contained in the macro-interpreter, characterizing the logical evolution of interpretation.

This information also, may be put in three parts :

- Program context
  . IC : instruction counter of the program in IML
  . CPT : address of end of block under examination
  . STACK II and TOP 2 ; address stack for the end of the included block and its pointer
- Interpretation context
  . AR : address register on interpretation program
  . STACK 2 : return address stack at the end of the decoding submicroprogram
- State of communication with the micromachine
  . Generated actions queue and its pointers
  . Queue of parameters to be transmitted and its pointers.

6.3. Micromachine context

The volume of significant context in the micromachine has been reduced considerably by the fact that the interrupts are enabled between two actions, as we have said above.

The information to be saved are the five registers making up the external register of the CU 2903. These registers are used to transmit the parameters between the various actions. It is to be noted that as this extension is in direct access with the descriptor memory, its content is saved in a single memory cycle.

This information will, therefore, be saved in the space descriptor of the interrupted process.

CONCLUSION

The high level computer architectures previously studied or realized concerned monotask languages. This study shows the principal problems met in the implementation of a multi-task real time language.

Interpretation processing has been divided into three classes :

- organic processing associated with the management of a multi-task system
- formal processing associated with the control of one task
- effective processing associated with the execution of each instructions of one procedure.

The hardware structure has been designed to support efficiently these three kinds of processing.

The realization of a prototype able to support the LTR language should allow the validation of these concepts.

REFERENCES

[1] LTR - Manuel de Référence 5616/U/FR

[2] LTR - Manuel d'Implémentation 8072/U/FR

[3] LTR - Manuel d'Utilisation 5618/U1/FR
CIMSA 10-12 Avenue de l'Europe 78140 VELIZY France

[4] J.PETIT, D.LITAIZE, B.LECUSSAN, J.P.SANSONNET
4.1. A microprogramming strategy for HLL interpretation SIGMICRO NEWSLETTER Dec.76 Vol.7 n°4
4.2. An efficient hardware tool for bit pattern manipulation 2nd Symposium on Micro Architecture EUROMICRO - 1976 - Venise

[5] Y.CHU
Concepts of a high level language computer architecture - Proc.ACM Conf.Minneapolis,MN, pp.6-13 - Octobre 1975

[6] Y.CHU
Direct-execution computer architecture
Proc.IFIP Congress,Toronto,Canada,Aug.1977

[7] Y.CHU
Issues an concepts of high-level computer architecture - IEEE Computer Society, 1979

[8] Symposium on High Level Language Computer Architecture ACM.IEEE Nov.7-8,1973 Univ.of Maryland
8.1. W.C.NIELSEN
Design of an aerospace computer for direct HOL execution
8.2. N.M.BLOOM
Structure of a direct high level language processor
8.3. GLOSS
A high level language machine
8.4. L.N. Mc MAHAN et E.A.FEUSTEL
Implementation of a tagged architecture for block structured languages
8.5. V.R. BASILI et A.D.TURNER
A hierarchical machine model for the semantics of programming languages

[9] L.W.HOEVEL
"IDEAL" directly executed languages : an analytical argument for emulation
IEEE Trans.on Computer, August 1974

[10] J.C.STRAUSS et K.I.THURBER
A computer design for real time command an control - EASCON 76

[11] K.I.THURBER and al.
A computer architecture for an advanced real time processing system - COMPCON 76 EAST

[12] A.S.TANENBAUM
Implications of structured programming for machine Architecture - Comm.ACM March 78 Vol.21 n°3

[13] W.T.WILNER
Design of the Burroughs B1700 - Proc.AFIPS FJCC Vol.41 AFIPS Press Montvale 1972 p.489-497

[14] W.T.WILNER Burroughs B1700 Memory Utilization Proc.AFIPS FJCC Vol.41 AFIPS Press Montvale 1972 p.579-586

An Architecture for the Dynamic Optimization of High-Level Language Programs

Samuel P. Harbison
Wm. A. Wulf

Carnegie-Mellon University
Department of Computer Science

Abstract. We introduce an architecture which performs many of the optimizations commonly seen in sophisticated compilers for high-level languages, including redundant expression elimination and the movement of invariant expressions out of loops. The instruction set of this machine allows simple compilers to produce a graph-structured object code which is both compact and efficient. The architecture features a cache which records the values and dependencies of HLL expressions in order to avoid later recomputations and memory references. Preliminary experimental results indicate a speedup approaching a factor of two over a pure stack architecture on some programs.

1. Introduction

The arguments in favor of closing the "semantic gap" between source program and object program are well known by participants of this conference. Myers [1] characterizes the job of the computer architect as determining the proper division of total system functionality between software, firmware, and hardware. Two extremes of this division are possible. At one extreme we have traditional architectures which tend to leave too much to the software and are ill-suited to the software they execute. Complex operating systems are necessary to make them useful; complex compilers are necessary to make high-level languages (HLLs) execute efficiently. At the other extreme we have architectures which attempt to execute high-level languages directly. These architectures are often inefficient themselves; program representations appropriate for programmers are not always appropriate for computers. It is likely that better cost-performance can be achieved by an architecture which falls somewhere between these extremes. Our architecture is one of many such; it is aimed at reducing or eliminating the need (and hence the costs) of optimizing compilers by performing important optimizations in hardware. It does not directly address other dimensions of the problem, such as the complexity of operating systems.

The total cost of optimizing compilers is great. Their construction is a formidable software engineering task. The code they produce is almost always obscure, occasionally worse than no optimization, and sometimes just plain wrong. They also execute more slowly, and hence exact a price on each compilation. Research is underway in several places aimed at reducing this cost through the automatic, or semi-automatic, generation of such compilers [2]. Our approach to this problem is different; we are trying to raise the hardware/software interface *above* the level of the compiler's optimization phase, thus reducing the compiler's task to (mainly) lexical analysis and parsing. Efficient algorithms for these phases are known, and the automatic construction of such compilers would be within our grasp.

Our architecture is able to perform two common and important optimizations: redundant expression elimination and a type of code motion typified by the movement of invariant expressions out of loops. These optimizations traditionally require sophisticated flow analysis during compilation, so their elimination from compilers should be beneficial. Our research is aimed at determining how big an impact this architecture can have on the total cost-performance of a compiler-architecture pair.

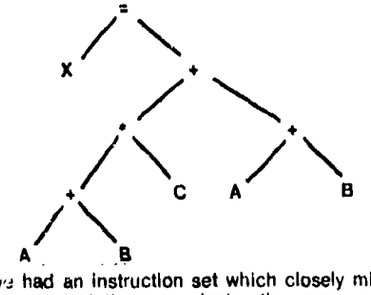
In this paper we will introduce the architecture and argue its advantages informally and by example. Other work is under way to determine the architecture's quantitative benefits over a range of real programs. Because we are interested in basic feasibility, we defer the specification of many details which would be necessary before the architecture could be realized. In particular, we are not specifying how to implement the architecture, nor are we specifying the instruction set beyond what we absolutely need. So as not to be overly distracted by language issues, we have chosen FORTRAN as our high-level language. We believe that the necessary extensions for other languages would be no more difficult on our architecture than on others, and therefore they are irrelevant to the current goals of the research.

2. Basic Concepts

To briefly outline the thrust of the architecture, consider the FORTRAN statement

$$X = (A+B)*C + (A+B)$$

which has this parse tree:



Suppose we had an instruction set which closely mimiced this parse tree representation, one instruction per node. Each instruction might be a triple

[OPCODE, LEFT-PART, RIGHT PART]

where LEFT-PART and RIGHT-PART would be addresses of instructions which calculate the operands. The execution of an instruction would consist of recursively evaluating the left- and right- parts of the instruction, followed by the application of the indicated operation. This architecture could be implemented using two stacks: one to hold intermediate computations and one to hold partially-evaluated instructions during the post-order traversal of the parse tree. The order of instructions in memory would be irrelevant in this instruction

set--the control flow is specified explicitly. The translation of the above statement would be

```
        *,X,P1
P1:     +,P2,P3
P2:     *,P4,C
P3:     +,A,B
P4:     +,A,B
```

This instruction set is obviously very inefficient, but it can illustrate two points. First, because the instructions labeled P3 and P4 are identical, there is no reason to duplicate them; we can eliminate P4 and change P2 to

```
P2:     *,P3,C.
```

The subexpressions giving rise to P2 and P4 are called, in the parlance of compilers, *formally identical* or *congruent*. This simply means that they are identical in form--not necessarily that they have the same value. It is both simple and efficient to detect formal identity during parsing, and doing so at compile time allows us to represent programs more space-efficiently in our architecture. By contrast, detecting common subexpressions, i.e., formally identical expressions that also are guaranteed to have the same value at execution time, is not as simple or efficient. Our architecture will not require the compiler to do this.

Notice that even though the expression "A+B" is represented only once in the object program (using the aforementioned compaction), it is actually evaluated twice in the implied traversal of the parse tree. The structure of the object code gives us the possibility of avoiding this recomputation. Suppose that after completing the evaluation of P3 (while computing the LEFT-PART of P1) we saved the "value" of this instruction in a cache, labeled by the address P3. If we checked that cache before evaluating each instruction operand, we could retrieve the value of P3 when computing the RIGHT-PART of P1 without actually recomputing it. Suitable care would have to be taken to record dependency information in the cache so that we could remove the value, should either A or B change in the future.

Our architecture provides such a cache, which is the major source of execution-time efficiency. The effect of using this cache corresponds closely to the elimination of redundant expressions by optimizing compilers. In fact, this technique may be superior, because it can eliminate expressions which are redundant under the particular execution history of the program. Consider, for instance, the following FORTRAN statements:

```
Y = A+B
IF (Y .LT. 0) A = A+1
X = A+B
```

Because the two occurences of "A+B" are formally identical, they can be computed by a single instruction which is referenced in two assignment statements. It can be seen that the value of the expression A+B, computed in the first statement, can remain in the cache unless the assignment to A *actually* takes place (invalidating A+B). The same mechanism serves to move invariant expressions out of loops, since any expression which does not depend on a value changed in the loop will remain in the cache.

This simple example illustrates our architectural goal: to provide an instruction set which preserves the structure of the parse tree in a way that permits both space-efficient representation (by having only one copy of the code for formally identical expressions) and time-efficient execution (by detecting and avoiding the re-evaluation of expressions whose value has not changed).

3. The Architecture

We now introduce the architecture and instruction set currently being used in our research. We would like to emphasize that this version of the architecture is a research vehicle--one intended (only) to test the feasibility of the ideas and their impact on performance. A realistic implementation would need to address other issues and would require careful tuning and elaboration of the instruction set.

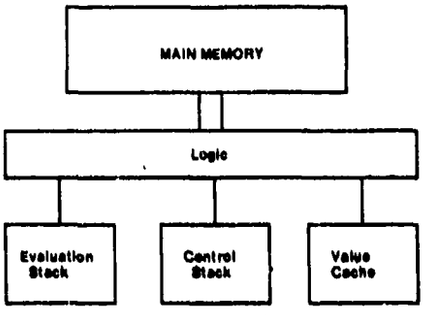


Figure 1: Major architectural components

There are four important parts of the machine, as indicated in Figure 1:

Memory	A linear vector of fixed-size words, indexed by address.
Evaluation Stack	A LIFO stack of words, used to hold intermediate values during computation, much the same as in other stack-oriented machines.
Control Stack	A LIFO stack of control information, used to control the recursive descent through the parse tree graph.
Value Cache	An associative memory used to save the values of expressions.

The Control Stack and the Value Cache will be explained in more detail later.

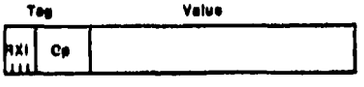


Figure 2: Memory word format

Every word in memory is a one-operand instruction, formatted as a [TAG, VALUE] pair (Figure 2). Even words usually thought of as data are, in this machine, instructions. The TAG field is further divided into a number of subfields, named R, X, I, and OP. OP is the operation code (e.g. ADD), and R, X, and I are single-bit fields denoting *Return*, *indeX*, and *Indirect*. (These will be described later.) The actual bitwise packing of these fields into a word is not too important, but for concreteness, we think of TAG as being 8 bits and VALUE as being (say) 24 bits. This would give us a 5-bit operation code and leave 24 bits for data or an address.

3.1 Instruction Classes

The instructions are divided into three classes according to how their operands are interpreted. The three classes are *data instructions*, *address-operand instructions*, and *value-operand instructions*.

Data instructions. The INT, REAL, and ADDR instructions correspond to the three data types recognized by this simple version of the architecture. Executing any of these instructions causes them to push themselves (VALUE and TAG) onto the Evaluation Stack, setting R=1 and X=I=0. The contents of the VALUE field in data instructions is the actual data (i.e., in INT instructions, VALUE is the integer datum, in REAL it is the floating-point representation, and in ADDR instructions it is an address).

The data instructions are quite like "tagged" data in other HLL architectures. In particular, we will assume automatic

type conversion throughout--there will not be separate instructions for floating-point addition and integer addition, for instance.

If X is a variable of type REAL with value 43.5, the name X will be bound to the address of a word containing (the instruction)

```
REAL 43.5.
```

The reason we make data words executable will become clear when the operand-fetching mechanism is examined later.

Address-operand instructions. These instructions include INC1 (increment-by-one), INC (general increment), STO (store), and the twelve conditional-jump instructions. In each case, the VALUE field is interpreted as an address, and this address is the instruction operand. The semantics of the instructions are as follows:

- STO — Removes the top word from the Evaluation Stack and stores it at the operand address. The R field is set to 1 in the stored word, and the X and I fields are set to 0.
- INC — Removes the top word from the Evaluation Stack and adds it to the word at the operand address.
- INC1 — Increments the value of the word at the operand address by one.
- JLT, JLE, JGT, JGE, JEQ, JNE — Remove the top value from the Evaluation Stack and branch to the operand address if the value is less than, less than or equal to, greater than, greater than or equal to, equal to, or not equal to zero, respectively.

Value-operand instructions. These instructions include PUSH and the arithmetic instructions, ADD, SUB, MUL, and DIV. For these instructions, the VALUE field is again interpreted as an address, but the operand is obtained by *evaluating* the address, as explained below. Otherwise the semantics of the instruction are as follows:

- PUSH — pushes its operand onto the Evaluation Stack.
- NEG — negates its operand before pushing it.
- ADD — removes the top word from the Evaluation Stack and adds it to its operand, leaving the sum on the Evaluation Stack. Type conversions are performed, if necessary, according to standard FORTRAN conventions. (Type information is available in the TAG fields of the data on the Evaluation Stack.)
- SUB, MUL, DIV work like ADD, with the left-hand argument being on the stack and the right-hand argument being the operand of the instruction.

Occasionally, one will want an instruction such as ADD to take *both* its operands from the stack. We therefore adopt the convention that if VALUE = 0, the operand normally specified in the instruction will be found as the topmost element on the Evaluation Stack. This applies to both address-operand and value-operand instructions.

3.2 Operand Evaluation

As stated above, value-operand instructions obtain their operands by *evaluating* the address which appears in the instruction. In this architecture, the evaluation mechanism uniformly replaces the "fetch-the-contents-of" mechanism in traditional architectures. To evaluate an address *A*, the current instruction-execution state is saved on the Control Stack and execution begins at *A*. After each instruction completes, the R bit is examined; if R = 1, the Control Stack is popped, terminating the new instruction sequence and returning to the previous one at the point where it was interrupted. In our examples, we will indicate that an instruction has R = 1 by appending "r" to the operation name.

Strictly speaking, there is no restriction on what instructions can occur in the new instruction sequence. However, it is our intent that the sequence of instructions, which is called a *phrase*, will leave a single value on the Evaluation Stack. If we make the further assumption that the computation is independent of data already on the Evaluation Stack, it is possible to speak of the *value of A*, or the *value of the phrase A*.

Note that a single data instruction, with R = 1, satisfies these conditions for a phrase. Hence, a single data word may be "fetched" by evaluating (executing) it.

3.3 Indexing

The X field is provided in TAGS to perform some simple address arithmetic. When X = 1, the address in the instruction is first incremented by the value found on top of the Evaluation Stack (which is removed as a side effect). The new address becomes the operand (for address-operand instructions) or the address to be evaluated to obtain the operand (for value-operand instructions). In our examples, we will indicate that X = 1 in an instruction by appending "x" to the instruction name, as in "STOx A".

Occasionally it will be useful to obtain an indexed address on the stack without evaluating the result. We therefore allow the X field to be set in the ADDR instruction, in which case the address present in the VALUE field of the ADDR instruction is incremented by the value on top of the Evaluation Stack, and the resulting address is pushed onto the stack.

3.4 Indirection

The I field is used to provide an extra level of evaluation in obtaining operands. When I = 1, the operand obtained by the above mechanisms is evaluated an extra time to obtain the true operand. For instance, in "STOI A", the address A is evaluated, and the actual store occurs to the address returned by the phrase A. In "ADDI A," the address A is first evaluated normally; then the resulting value of A is evaluated, yielding the operand.

This mechanism makes several assumptions. In particular, in value-operand instructions it is assumed that the value returned by the first evaluation is an address (so that it can be evaluated again). Likewise, in address-operand instructions it is assumed that the evaluation (the one caused by I = 1 is the only one) produces a value of address type.

When I = X = 1, the indexing operation is applied before the (first) evaluation.

3.5 Discussion

Returning to our original example, we can see what the code actually looks like in this architecture.

X = (A + B)*C + (A + B)

```
     PUSH    P3
     MUL     C
     ADD     P3
     STO     X
     - - -
P3:  PUSH    A
     ADDr    B
A:   REALr   23.5
B:   REALr   -3.0
C:   REALr   4.66E1
X:   REALr   0.0
```

Note how the evaluation mechanism is exploited in collecting the formally identical expressions into a single phrase (P3).

The indexing and indirection mechanisms are optimizations designed to facilitate address computations in array and structure accesses, much like the use of index registers in conventional architectures. In (a), below, we see the simplest form of indexing; in (b) the two occurences of "C(I)" have been implemented as a single phrase; in (c) the phrase has

been constructed to compute the address of C(I) since both the address and value are needed.

C(I) = A(J)

```
PUSH  J
PUSHx A-1
PUSH  I
STOx  C-1
- - -
```

(a)

X = (C(I) + B)*C(I)

```
      PUSH  L
      ADD   B
      MUL   L
      STO   X
      - - -
L:    PUSH  I
      PUSHxr C-1
```

(b)

C(I) = C(I) + B

```
      PUSHi L
      ADD   B
      STOi  L
      - - -
L:    PUSH  I
      ADDRxr C-1
```

(c)

These examples indicate that there is some choice in how to structure the object code. In terms of space-efficiency, any expression appearing in the source program more than once should be expanded as a separate phrase. Execution-time efficiency can be gained by additionally separating expressions used within a loop; if their value does not change, the effect is the same as if the compiler had moved them outside the loop.

3.6 The Value Cache

The Value Cache is the most unique and important part of the architecture. Its purpose is to save the value of phrases. Every time an evaluation is attempted, the Value Cache is first checked to see if it contains the phrase's value; if found, the value can be immediately entered on the Evaluation Stack without any need to actually execute the phrase in question. If the Value Cache does not contain the desired value, evaluation proceeds normally and the new value is copied into the Value Cache as a side-effect of the processing of the R field in the last instruction of the phrase.

An important part of the cacheing mechanism is keeping track of dependency information. The value of a phrase can depend on an unbounded set of memory locations--namely all those which are referenced in the course of its evaluation. Should any of these locations be changed, the old value in the Value Cache must be purged.

Because the space available to represent dependency information in the cache will be limited, we must have a way to encode the dependency information. A possible implementation is to represent the dependency set as a bit vector of length *n*. A dependency on a particular memory word with address *A* could then be mapped into one of the *n* bits by an operation on the word's address, *D(A)*. An inclusive "OR" of all encoded addresses would then represent the dependencies of the phrase. Purging from the cache all values dependent on address *B* could be accomplished by eliminating all entries which included bit *D(B)* in their dependency mask.

To explain how the Value Cache is used, we need some information about both the Value Cache and the Control Stack. The Value Cache is an associative memory, each entry of which has three fields:

VC·ADDRESS	address of phrase
VC·VALUE	value of phrase
VC·DEPENDENCY	dependency of phrase

Control Stack entries also have three fields:

P·ADDRESS	address of phrase
I·STATE	current execution state
CS·DEPENDENCY	accumulating dependency

There are four activities which involve the evaluation mechanism and the Value Cache:

Beginning an evaluation. The Value Cache is checked to see if it contains the phrase's value; if so, the value is immediately entered onto the Evaluation Stack and the evaluation is considered complete; dependency information from the Value Cache (VC·DEPENDENCY) is added to the dependencies being accumulated for the current phrase (CS·DEPENDENCY). If the phrase is not found, the current execution state is saved on the Control Stack and a new frame is added for the new phrase, whose evaluation begins. CS·DEPENDENCY for the new phrase is initially null.

During evaluation. Every execution of a data instruction represents a dependency; the dependency is derived from the address of the data instruction. The encoded dependency is added to the dependencies already recorded in CS·DEPENDENCY.

After evaluation. When an instruction with R = 1 is completed, the phrase value (the top value on the Evaluation Stack), P·ADDRESS, and CS·DEPENDENCY are sent to the Value Cache for recording as VC·VALUE, VC·ADDRESS, and VC·DEPENDENCY, respectively. (If the Value Cache is full, some mechanism for removing entries must be employed.) The Control Stack is then popped to return to the previous phrase; the dependencies of the completed phrase are added to the dependencies accumulating for the previous phrase. (That is, if phrase A invokes phrase B, phrase A's dependencies include those of phrase B.)

During a store operation. Whenever a STO, INC, or INC1 instruction is executed, every Value Cache entry which shows a dependency on the altered word is purged. (This may not be a perfect discrimination, depending on the encoding D(X).) The value being stored (itself a phrase) is entered into the Value Cache as a side-effect; its dependency is precisely itself.

As an example, consider the following (assume M(6) = 45):

K = M(I) + I

```
      PUSH   L
      STO    K
      - - -
L:    PUSH   I
      PUSHx  M-1
      ADDr   I
I:    INTr   6
K:    INTr   45
```

There are four phrases entered in the Value Cache after executing this statement:

VC·ADRESS	VC·VALUE	VC·DEPENDENCY
I	INT 6	D(I)
M+5	INT 45	D(M+5)
L	ADDR M+5	D(I) ∨ D(M+5)
K	INT 51	D(K)

If we later changed the value of I, the phrases I and L would be purged from the Value Cache, but M(6) (i.e. M+5) would remain, unless by chance D(I) = D(M+5).

4. Measurements

To obtain objective measures of the performance of this architecture, we resent here analyses of four simple programs: three production-quality statistical subroutines taken from the *Scientific Subroutine Package* and one simple quadratic-equation solver taken from an introductory programming text. When we say *production-quality*, we mean that there is no obvious way to rewrite the source program more efficiently in the statistical subroutines. In contrast to this, the quadratic-equation program contains several examples of formally identical (and redundant) expressions.

We examined the execution of these programs on three compiler/architecture pairs: on our architecture with a simple compiler performing no optimizations; on a DEC PDP-10 with the FORTRAN-10 optimizing compiler; and on a *modified stack architecture* (MSA). The MSA is a variant of our architecture, obtained by eliminating the evaluation mechanism (including Value Cache and Control Stack) in favor of the simple "fetch-

the-contents-of" mechanism; it is thus a simple stack architecture with the same one-operand instructions as in our architecture. The compiler for this architecture is identical to the one for our principle architecture.

Code size statistics were obtained from listings of the compiled assembly code. Execution statistics were obtained from instruction traces on the PDP-10 and from emulators of the other architectures. In emulating our architecture, we used a Value Cache with 100 entries and a 32-bit-wide dependency field with D(A) = A mod 32.

In comparing program sizes, we assume that a "word" is equivalent on the different architectures. Likewise, execution statistics are expressed as the number of memory fetches and stores (instructions plus data). We do not count internal processing, so all instructions take unit time unless they involve a fetch or store from memory. (We do not consider the Value Cache to be memory in this sense.) With this in mind, we present the data in Tables 1 and 2. Tables 3 and 4 present the same data as a fraction of the MSA values.

	Architecture		
Program	PDP-10	Ours	MSA
S1	186	211	224
S2	148	168	165
S3	80	94	96
S4	121	118	169

Table 1: Code size (words)

	Architecture		
Program	PDP-10	Ours	MSA
S1	2,162	2,414	3,647
S2	1,282	1,726	2,219
S3	6,516	9,666	12,942
S4	408	447	824

Table 2: Execution speed (fetches)

	Architecture	
Program	PDP-10	Ours
S1	.83	.94
S2	.90	1.02
S3	.98	.83
S4	.72	.70

Table 3: Code size (fraction of MSA)

	Architecture	
Program	PDP-10	Ours
S1	.59	.66
S2	.58	.78
S3	.50	.75
S4	.50	.54

Table 4: Execution speed (fraction of MSA)

The PDP-10 and MSA are in a sense upper and lower bounds for comparison purposes. The PDP-10 is a mature instruction set in the traditional Von Neumann mold; it has been carefully designed and optimized. MSA on the other hand is the simplest stack machine one can imagine. Likewise the PDP-10 incorporates a sophisticated compiler, whereas the other architectures have very simple compilers. (In particular, they do not even have to do register allocation.)

The data confirms that the PDP-10 is still the more highly optimized architecture, but in the case of the S4 program, our simple compiler was able to produce code which was more compact and which executed almost as quickly. Clearly the benefits depend to some extent on the degree to which redundant expressions can be eliminated during compilation and execution. However, even with well-coded programs, we see a significant improvement over a simple stack architecture.

Of course, these few examples cannot alone establish the benefits of our architecture. It is meant only as an informal argument to establish the *possibility* of such benefits, even in programs not easily optimized. We hope to provide more quantitative evidence on a wider range of programs in the future, along with more information on the effect of the size of the Value Cache and on cache-full policies [3].

References

[1] Myers, G. *Advances in Computer Architecture*. Wiley, 1978.

[2] Leverett, et al., An Overview of the Production Quality Compiler-Compiler Project. Technical Report CMU-CS-79-105, Carnegie-Mellon University Computer Science Department. 1979.

[3] Harbison, S. P., The Dynamic Optimization of High-Level Language Programs. Ph.D. Thesis, Carnegie-Mellon University Computer Science Department. To appear.

ARCHITECTURAL SUPPORT FOR ABSTRACTION

J.K. Iliffe

International Computers Limited (*)

Abstract

A mechanism for supporting fine-grain program protection and abstraction in a multi-computer context is described. It is argued that such features are necessary to support high level user interfaces and particularly high level language implementations using microprogram control, but that their cost must be small in relation to microinstructions. The mechanism is currently being investigated by simulation techniques as part of a general-purpose system study.

Objectives

The most important objective of general-purpose computer design is to model accurately, reliably and efficiently the data of widely varying problem domains. We might instance records, messages, tax tables or graphical images as typical classes of data familiar to computer users, and to the extent that the attributes of a class, neither more nor less, are recognised we can say that a successful *abstraction* has been achieved. We define a 'high level' architecture as one that supports such abstractions for an open-ended list of classes. Its importance is that it enables complex data processing applications to be developed and maintained in a reliable state by offering to information engineers something comparable with the subassemblies and precise tolerances of, say, mechanical design. Overall, one expects as a result to produce better systems more quickly and more reliably and at a lower cost than would otherwise be possible.

The complexities of operating systems have drawn attention to the importance of program structure, most designers making use of the ideas of *task* (i.e. process), *file*, *segment*, *event* and others in abstract form. We could include *code segment* in the list and thus lead to the accurate, reliable and efficient modelling of high level languages, but it would be a mistake in the present context to put either operating system or language engineers in positions of privilege since (through no fault of their own) that seems to guarantee poor response to user requirements. For example, in range-defined architecture (in the style of the IBM 360) the micro programmer has in effect been a language engineer with considerable privilege: for precisely that reason it has been impractical to make wide use of improvements in the encoding of high level languages which depend on having variable intermediate code formats. Attempts to define architectures at even higher level run correspondingly higher risks.

The order of events, therefore, is to define the abstraction mechanism first and then use it to model whatever operational behaviour is required. But what is meant by doing that 'efficiently'? Fifteen years ago, under the umbrella provided by the IBM 360, it seemed sufficient to achieve the objective with 'no increase in program size or loss of speed', which is essentiallt what happened with the Basic Language Machine[1]. Today that umbrella is permeable and to out-perform current range-defined architectures is commonplace. The essential requirement now seems to be to provide the benefits of abstraction at the finest level of description used by system, language or application engineers - in other words at what is usually regarded as the microcode level. Once that is done, the way is open to realising in a practical context the advantages of microcoding that have often been demonstrated under special conditions.

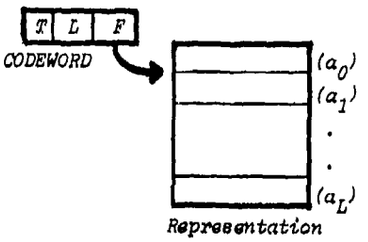
In this paper I shall outline a design, which for reasons soon to become clear is called a "Pointer-Number system", which demonstrates one way of meeting the objectives. It takes account of system requirements not mentioned here, and has been carried to a detailed simulation in order to make realistic performance estimates. In the next subsection we review the techniques on which it is based and the range of problems that have to be solved at the next stage of design. The following subsections outline respectively the 'PN Machine' and 'PN System'. Finally, some conclusions are drawn from the experimental work done so far. The reader is referred to the *PN System Manual*[2] for more detailed explanation and justification.

Abstraction Mechanisms

The basic requirement is to mechanise the ideas that might be expressed as: "Let A be a class of objects with attributes $\{a_i\}$ $i = 0 \,..\, L$", "Let x be a (member of the class) A", "Let y

(*) Present address: Department of Computer Science and statistics, Queen Mary College, University of London

denote (the same member of the same class as) x", and so on, all the representations being within the limits of a finite computer store. In programming terms this quickly resolves into the use of descriptors or pointers as a type of operand distinct from the attribute sets that represent the individual objects, a construct that has been used from the earliest days, though it was not precisely engineered until segmented storage came into use in the early 1960's (Figure 1). In the case of program space the connection between (indexed) pointer and attribute is notionally direct, but it is a simple extension of the same idea to interpret the descriptor as referring to a member of any given class of objects, which was the generalisation made in the Basic Language Machine (Figure 2). In the



T: Segment type
L: Maximum index
F: Location

Figure 1: Storage segment

latter case the pointer contains indices c, id that uniquely identify the class and object in question. In accordance with current practice we refer to pointers used in this indirect way as "capabilities" but the term "codeword" is retained for the special case of reference to storage.

It is implicit that pointers cannot be forged, otherwise the whole point of having precisely engineered program structures is lost. On the other hand they must be manufactured somewhere and the class manager must be able to manipulate the representations directly. Such considerations lead to the notion of protected domains characterised by sets of pointers that define the 'rights' of a program at any instant of its execution. As control flows from one domain to another there must be correspoding changes in the list of rights.

Before discussing possible mechanisations we should be aware of the performance parameters to look for in the final analysis. Amongst the most important is the time taken to access the attribute given a valid pointer: there is no absolute figure to aim for, but it is required to be short in relation to the class of operations that it supports. For example, in dealing with files or tasks the individual operations are fairly substantial and a number of capability systems have been implemented in which pointers are interpreted by the operating system without serious loss of speed. In moving towards simpler operations the interpretive mechanism must be refined and assisted, first by microprogram and finally by hardware, and in the present context the stringent requirement of having low overhead in relation to micro-operations forces us to disregard all but the most delicate controls. In the model provided by Figure 1 we might nominate the 'effective storage access time' as the relevant parameter. In Figure 2 the critical time is that taken to move the locus of control from the 'user domain', containing the capability, to the 'class manager domain' in which interpretation takes place and back again. In either case, if the observed cost is too high users will tend to avoid the facility and lose its benefits.

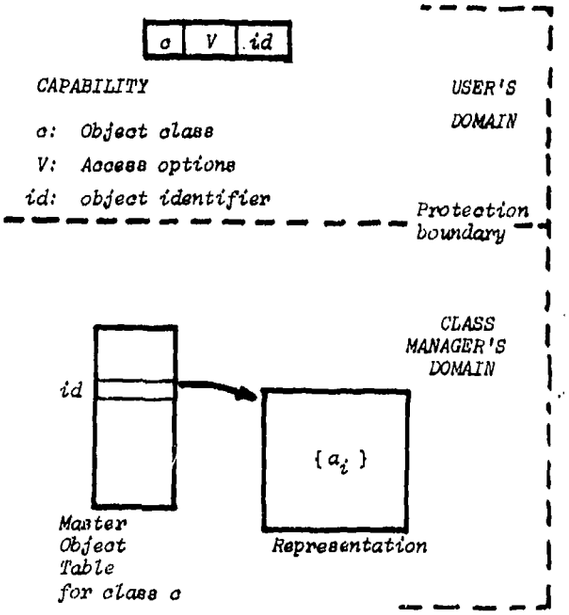


Figure 2: Indirect class representation

The other factors are more difficult to quantify because they entail the inevitable compromise between cost of management and ease of use. It might be asked: "If members of a class are generated at a given rate, what is the resulting management overhead?". For example, how often can one open new files, create messages, or assign new tasks without undue penalty? Clearly, some costs are passed on to storage management which has to provide file control blocks, buffers, task vectors and so on, but there remains the responsibility for master object tables, for recovering 'dead' identifiers, and for error management. The techniques available for reducing costs are mainly concerned with the time taken to scan the program space looking for particular classes of pointer and might be aimed at eliminating that need

entirely, e.g. by:

(a) enlarging the master object tables to service all foreseeable requests; or
(b) restricting the use of pointers, e.g. by indirect reference through system tables or by linguistic devices;

alternatively we can seek to minimise the actual scanning time by:

(c) limiting the extent of pointer-bearing segments; or
(d) constraining the program structure, e.g. to separate task domains or to a 'tree' form.

In any well-designed capability system the constraints are small in relation to the benefits they bring, but the fact remains they are a psychological hindrance to widespread acceptance. The best way round that, architecturally speaking, is by:

(e) providing high speed memory scanning and updating operations, enabling many of the restrictions to be relaxed.

The last solution is pursued in the PN system by using what are effectively microprogrammed management procedures in conjunction with hard-wired 'planar' memory scanning functions.

Returning to the primary measure of storage access rate, it is clear that no scheme dependent on validating pointers at time of *use* (against access list, segment table, capability registers, etc) would be acceptable, and in order to compete with 'unrestricted' access mechanisms we are forced (i) to admit pointers as operands used directly by machine instructions; and (ii) to control their *formation* so as to preserve the integrity of programs. There still seems to be no better way of doing that than by using a tagged register format. However, in moving the control mechanism to microinstruction level the interpretation of tags must be resolved in single microorders. In theory, just one tag bit is necessary, to distinguish between *pointers* and *numbers*, but it will be seen in the next subsection that fifteen pointers and one form of number are distinguished by a four-bit tag code.

We have already seen that because of its practical importance storage is distinguished from all other abstract classes. A further distinction is drawn between sharable (*global*) and unsharable (*local*) data areas. The corresponding pointers are *codewords* and *addresses* respectively, which have almost identical properties in normal use. It is unfortunate to make the distinction, but it reflects the fact that controlled access to shared resources uses a single level of indirection which is otherwise unnecessary. The same mechanism is used to distinguish between data that might be at a remote site in a multicomputer system (and therefore 'global') and data areas that are strictly local.

Figure 3 illustrates the use of pointers in referring to different program workspaces. The transformation α is handled by capability managers, while β is the responsibility of the segment manager. Parallels can be drawn between writing in a conventional high level programming language and operating on global data, between microprogramming and working at local level. However, a key feature of the PN system is that sharp distinctions are not drawn and it is easy to move from one level to the next.

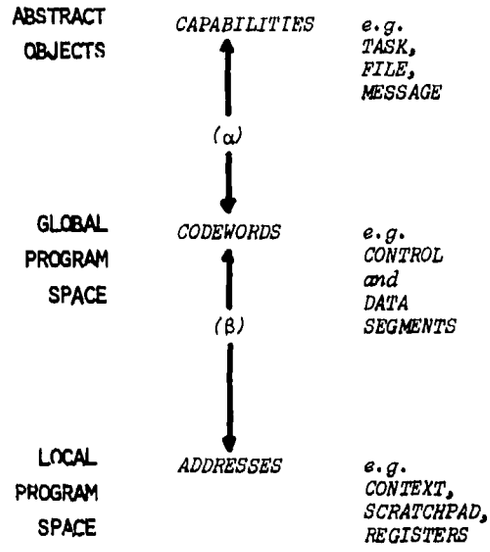


Figure 3: Levels of program space

A protection domain is defined by the combined effect of two sets of rules: those that govern the *inheritance* of access rights in registers and storage, formation of new addresses from old ones, restriction of access options in capabilities, etc, all of which are reductive in character; and those concerned with the expansion of rights in passing from one domain to another. The ability to expand rights depends on some prior authority saying in advance that "program module *M* shall only access resources m_1, m_2, ... m_k", which in turn devolves on the *construction* of control segments and associated data. Apart from the need for speed and flexibility in implementing such a rule we also require that it should be easy to apply and not expensive to support. In the PN system the region into which rights expand is defined by a set of resources known as a *base*. There will be several bases in a system, so there is scope for partitioning at that level. The objects m_1, m_2, ... m_k are identified by indices that are embedded in object code. That seems to be the most economical way of changing access lists, since it

is done at zero cost in conjunction with control transfers. It will be shown later how the interconnection mechanism is supported by machine functions in the context of a dynamically changing base, task and module population.

Pointer-Number Machines

In order to evaluate the above ideas in a practical system context a detailed machine model known as "microPN" has been defined and simulated. The intention has been to provide full support for abstraction in the context of an assembly of processor-memory pairs, each comparable in cost and speed with current microprogrammable machines.

The main components of microPN are shown in Figure 4. The register file (X) consists of 16 32-bit general-purpose registers. Most internal machine operations can be completed in one or two ALU cycles, typically processing the 'high' halves of the operands first, which includes checking their tags, followed by the 'low' portions. The ALU carries out elementary arithmetic, logic and shift operations on numeric words, and the special operations required in controlled pointer formation.

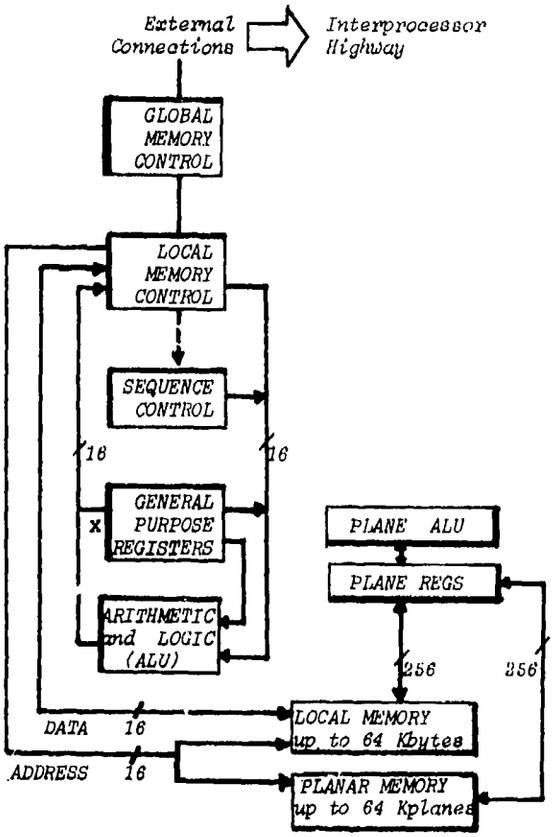


Figure 4: General schematic of microPN machine

The sequence controller plays a conventional role. The most frequently used control fields (control pointer, condition codes) are held as separate registers, the remainder being found in the general register file and protected from mis-use by overall controls on program construction. They include base and task indices, stack base and current stack frame, current control segment index.

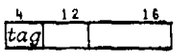
The local memory controller serves requests for data and instruction accesses within the processor and external requests arriving via the global memory controller. The memory operations include normal fetch and store of byte, word and tagged values, and 'planar' accesses arising from the use of local memory as an active storage device.

The four high order bits of each register conatin a tag, as shown in Table 1. The remaining 26 bits are interpreted accordingly. The format of tagged elements in store is the same as for registers. Note that tags 0..7 are 'global', and have the same meaning for every machine in an assembly, while tags 8..f are addresses with no meaning outside the processor in which they occur.

TABLE 1

microPN REGISTER FORMATS.

GLOBAL OBJECTS



Arithmetic	0	high	i	Integer
	1	s	i	Entry pointer
	2	c	id	Indexable capability
	3	s	i	Indexable codeword
Non-arithmetic	4	c	id	System capability
	5	s	i	Control pointer
	6	c	id	Capability
	7	s	i	Codeword

LOCAL OBJECTS

Read-write	8	L	F	Byte sequence
	9	L	F	Word sequence
	10	L	P	Plane sequence
	11	L	F	Tagged sequence
Read-only	12	L	F	Byte sequence
	13	L	F	Word sequence
	14	L	P	Plane sequence
	15	L	F	Tagged sequence

It can be seen from Table 1 that capabilities and codewords have 'arithmetic' and 'non-arithmetic' forms: in the former the object index or identifier can be altered by arithmetic operations. In neither case can the class or segment index be changed without authority. A distinction can thus be drawn between a 'singular' reference to an object or element of a segment and one that can be treated as one of a sequence.

Local objects are the addresses in local memory (starting at byte position F or plane P) of $L+1$ consecutive elements of the specified type. The local store is extended by an optional planar store which serves as a back-up for the (presumed) faster local memory. In microPN planes are just 256 bits in size, and to enjoy the full advantage of the addresssing scheme it is envisaged that planes of 1024 or 4096 bits will be used in practice. Data is transferred between levels via the planar register unit.

Global segments are addressed indirectly by the global memory controller through a segment table which might be associated with another micro-PN processor in the same assembly. Segment table entries have the same form as addresses. Figure 5 shows the principle of interprocessor communication assuming a bi-directional data and address bus of 32 bits. The requesting program applies a memory function m to the codeword (s,i) . From s the position of the 'host' is found; if not in the same processor-memory pair the parameters (m,s,i) are transmitted to the receiving module, where the function m is interpreted with reference to its segment table. A suitable reply is sent to the requesting program. Details of the interaction depend on performance objectives and cannot be meaningfully examined until program design strategies have been fully explored.

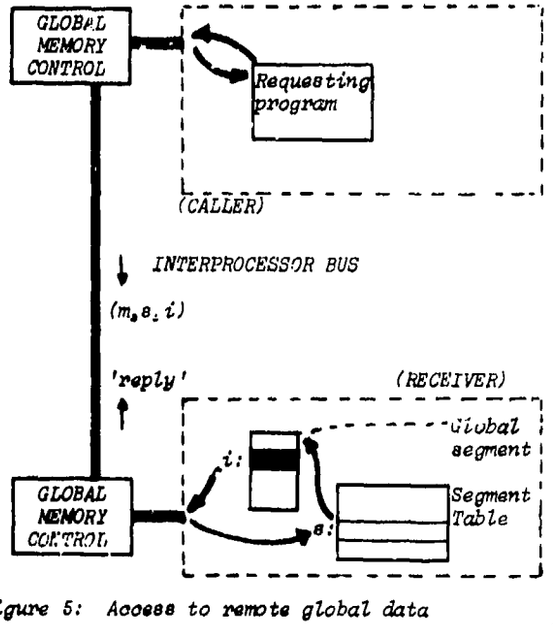


Figure 5: Access to remote global data

The 'plane ALU' operates on three planar registers, each 256 bits in microPN: an accumulator which can be regarded as 16 words of 16 bits or one bit from each of 256 words stored in plane sequence; a carry plane associated with the accumulator for bit-serial operations; and an activity plane that selectively controls store write operations. A further set of operations is provided to move the accumulator in either 'row' or 'column' direction, with linear or cyclic edge connections. The planar functions are designed primarily to assist in high speed operations on numerical data, digitised images, signal data, etc. However, in the present context planes play a prominent part as 32-byte units of memory allocation, and planar functions are used in module interconnection and scanning operations. The conventional store operations are extended to transmit numeric data between general purpose registers and word planes along common row or column data lines. Hence the design achieves another fundamental objective, of easy transition between 'parallel' and 'scalar' modes of operation.

In a tagged machine the instruction set is designed to carry out normal arithmetic and logical functions on numeric data and to provide separate functions for operating on pointers. Thus the 'modify' function in various forms applies to any address and increases F (or P) by a given amount, decreasing L accordingly. The 'limit' operations reset L to a lower value. If the bounds of the original sequence are exceeded an 'invalid address' (system capability class 8, see below) is returned. In that way the current protection domain can be delineated with a precision of one byte.

In microPN there are eight primary function groups, of which four are tag-independent and four restrict the tag of one or two general-purpose registers. The tag limitations can be simply expressed in tabular form and as far as can be seen would have very little effect on cost or speed. Nevertheless the essential protection mechanisms have been retained.

An incidental effect of the PN protection scheme is to make it easy to apply 'execute-only' options to control segments. Advantage has been taken of that to preserve some engineering flexibility and to undertake some security checks during program translation. For example, all register, base, label and system function indices are checked by the compiler and written into code sequences knowing that they cannot be changed by the user. Similarly, privileged function codes (such as 'set tag') can be generated without direct control by the programmer and there is no need for a distinct 'microsystem state'. There is, of course, the possibility of code being corrupted by store malfunction which, like pointer errors, could lead to wider breakdown. Whether to control such errors by further checks on the code, the pointers, the task space, the processor, ... or at some other boundary depends on the type of reliability and availability that is demanded.

Pointer-Number systems

The PN system supports ten classes of abstract objects, see Table 2. The aim of each abstraction is to disclose as much about each class as the user needs to know in order to operate on it efficiently, concealing attributes that are irrelevant or liable to change. For example, binary instruction formats are concealed in the definition of control segments in order to allow freedom to change the instruction coding. The system abstract objects constitute the resources available for program construction at the lowest design level. To reach the level of facility normally seen by application or system programmers new classes of object such as 'message' or'queue' will be implemented in terms of those that already exist. The use of separate tag codes for 'system' and 'user' capabilities, while not strictly necessary, is helpful in defining system structure.

TABLE 2

PN System capabilities

(All elements in this group have tag 4, the index value id identifies a member of the class o)

o:		
	0	*Null*
	1	*Control segment*
	2	*Pointer segment*
	3	*Base*
	4	*Task*
	5	*File*
	6	*Host (Processor-memory pair)*
	7	*CFC (see text)*
	8	*Function error*
	16	*Numeric segment*

The principles of capability management are widely understood, so we examine here only aspects peculiar to the PN system.

Data segments

An important distinction is drawn between data segments (identified by numeric or pointer capabilities) and access paths to them (identified by codewords). A given segment may be accessible through 0,1 or more such paths at a time, each using a distinct index. Their allocation is controlled by system functions to facilitate data sharing at global level. The distinction is important because not all operations on segments demand access to individual elements: for example, one might want to know the type or size, position in the hierarchy, or simply to pass the segment capability as a parameter.

Control segments

In the same way, control segment capabilities are distinguished from control pointers (tag 1 or 5). A control segment contains encoded instructions and data derived from definitions given in the system programming language. Although many features of the PN machine are abstracted the segment size, which contributes to channel loading and working set requirements, is not: in microPN the maximum size is 4096 bytes. There is only weak connection between segments and control flow, i.e. change of segment does not imply change of procedure, nor vice versa, the reason being that although one can sometimes take advantage of such conventions it is usually undesirable to couple logical control structure to physical store assignment.

The definition of control segments includes a precise specification of the registers they use, their entry points, and external connections that may be established with reference to the environment at time of use. The compiler, in conjunction with machine functions, ensures that the bounds so defined are strictly observed. That is the essential requirement of software engineering, brought down to 'micromachine' level. A logical property of a control segment (Figure 6) is that the only resources it can use are those defined in or accessible from the registers at a point of entry (e_0, e_1, or e_2 in Fig.6), or those acquired by expansion (m_1 or m_2), or those that it creates by using one of the resource managers.

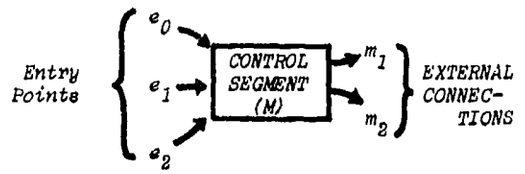


Figure 6: Interconnection of control segments

It is theoretically attractive to have precise control over which of the entry points to a module can be used in a given context. For example, if *M* controlled a class of queues and e_0 e_1 and e_2 allowed users to 'join', to 'leave', and to 'delete' a specified queue, it might be desirable to withhold e_2 from all but a limited subset of users. That would mean having distinct pointers for each entry point and increased overheads in the management of bases. On balance, it is preferable to define inly a single codeword for the module, say *M*, and to enumerate the entry pointers as *M*, *M+1*, and *M+2*, corresponding to e_0, e_1 and e_2 in the example. More precise control can be achieved by (a) using separate control segments for 'join' and 'leave' on one hand and 'delete' on the other;

(b) by using part of the identifier field to encode the permissible operations (the 'access options' in Fig.2); or (c) by controlling the indexing operations in a higher level language.

Once formed, a control segment is ready for execution. There is no need to load or consolidate it into a particular program, task or processor space. The reason for that design decision is that it gives the greatest flexibility in program construction at a cost which, from experience of similar systems, appears to be small. External connections are defined by reference to the current base and task, but since the *same* segment might be in concurrent execution with reference to several different bases and tasks, each with different components, the environmental vectors are treated as 'sparse' and connection is made by an associative search using the resource name as argument. The association is done by parallel (planar) operations and is relatively fast.

The only method of expanding rights is via the list of resource names, and strictly speaking the inclusion of a name in a control segment should be subject to formal checks. It would be possible to give a list of 'valid' names to each user or software design group, but here again the advantage gained from a strict rule of construction must be balanced against the cost of administering it. In our experience informal controls are sufficient for most applications, wherein the 'prior authority' can verify by inspection of the source code that a control module (such as *M*) cannot extend its effect beyond the permitted bounds (such as m_1 and m_2).

Function errors

For any machine or system function construed as 'failing' there is a choice of aborting the task or returning a recognisably invalid result from system capability class 8. The choice is a practical matter: for example, illegal tags abort the program, whereas address overflow returns an invalid address. If the former option is taken the 'result' of a task is itself a class 8 capability. In all cases the encoding of the index field gives the function type and reason for failure.

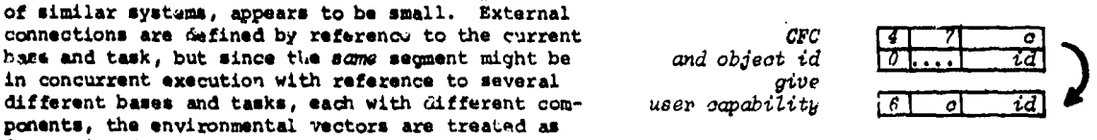
A similar convention can be applied in the user domain, returning class 0 system capabilities ('Null') to indicate failure. With regard to dynamic type checking, the user can easily 'break open' a capability to examine its class and tag fields. There are three courses of action:

(a) to assume all types are correct and expect to fail later (e.g. on tagcheck) if they are not;
(b) to check types and fail gracefully; or
(c) to check types and return a Null result.

There are many tactical variations; which to use depends on the level of understanding between caller and callee, and it is important not to preempt the decision in system design.

Capability management

To form a new class of abstract objects the designer requests permission from the system, which returns a capability-forming-capability (CFC) containing the index *c* of the new class. To form a new capability one can then present to the system that CFC together with the object index *id*. In return a tag 6 user capability, class *c*, index *id* is obtained:



We now see that the typical 'package' dealing with a class of objects consists of a manager *M*, whose name is made public, and essentially private data structures such as the master object table and CFC whose names (m_1 and m_2) are excluded from other segments. Disclosure of *M* will also document the functions of its entry points. The 'difficult' aspects of *M* are concerned with index management which, as we saw earlier, leads to various forms of evasion. In microPN, system support is offered to delete either (a) a given capability or (b) a capability class (authorised by the CFC) from program space.

Inevitably, pointers must be scanned looking for such capabilities. In a multicomputer system the rate of scanning store has two important characteristics: (i) it is relatively high, because of the close connection between processors and memories, and (ii) it is roughly constant because additional memory brings with it additional processing power. As a result we can suggest index management strategies based on the use of small m.o.t.'s whose entries are recycled when no longer in use.

For practical reasons store allocation is serviced by a special set of system functions, but the above comments on index management are equally applicable to codewords and addresses. The planar memory functions are particularly important in store compaction.

* * *

In summary, it might be said that the main problem of microsystem design is not to invent new facilities but to select a basic subset from the range of possibilities on offer. It is paradoxical that at a time of great abundance in hardware the need for stringency in design is greater than ever, but the fact remains that there are great dangers from 'overkill' in hardware and software. In microPN the decisive factors are the need to maintain security at microprogram level, and unwillingness to suffer loss of performance in doing so. Emphasis is therefore placed on the ability to *construct* high level systems rather than commit the design in one direction or another.

Simulation

The PN design is based on a computer module assumed to be comparable in speed and complexity with current microprogrammable machines. Besides playing its part as a member of an assembly, each must satisfy the most exacting requirements of program reliability and language implementation, which carry over (still unsatisfied) from conventional design. Before making specific hardware recommendations it is necessary to study in depth the program organisation and behaviour that can be expected in practice, so the approach has been to simulate one computer module and to make measurements from which the performance of an assembly can be inferred. The simulator runs under the UNIX operating system on the PDP-11 series of computers. Facilities available include a system implementation language, system support and error management functions, library and on-line documentation.

A multitask system is simulated, and between any two control points it is possible to count:

(i) instructions obeyed
(ii) local store accesses
(iii) global store accesses
(iv) stack usage
(v) procedure calls
(vi) module interconnections
(vii) planar functions obeye.
(viii) planar routing distance
and (ix) interrupts.

Elapsed time in the host system is also available, and PN system functions can readily be modified to give measures of resource usage, static measures of instruction coding, etc. The significance of the above figures should be clear. Taking store traffic as the main parameter of performance, it is found that for every 100 bytes of instruction about 30-60 further bytes of data are handled. If the data were all global, the overhead of segment table access would thus be 25-40%, but that is never the case: it is rare for less than 90% of data accesses to be local and we conclude that the overhead is negligible.

As already shown, change of access list is implicit in moving from one section of code to another, but for each register saved or restored at a domain boundary six bytes of data and instruction are used. A complete task change in microPN generates about 500 bytes of store traffic, while the search of external names associated with module interconnection generates about 200 bytes (50 instructions obeyed). In scanning operations, about one machine instruction is obeyed for each pointer examined, so that a typical stack (less than 100 tagged values) would be scanned in 10μsec.

The above figures begin to provide the context for high level program design decisions, e.g. whether to use global or local workspace, how to distribute segments across computer modules, when to use advanced forms of binding, what mixture of interpretive and in-line control to use, and so on. Quite often a high performance figure is traded for some other attribute such as resilience or responsiveness which is difficult to quantify. A vital objective is to achieve performance in convertible shape: that applies particularly to the levels of abstraction and control, because their interfaces undoubtedly decide whether what is possible in theory is actually achieved in a practical system.

Costs are equally difficult to quantify, and care must be taken to compare designs with similar facilities. For software engineering, controlled pointer formation is far more effective than segment/page table control and costs much less on a gate-for-gate basis. The most conspicuous cost of microPN is the 16*16 bit planar arithmetic unit, whose main contribution to the *system* is in memory management. Its use enables the dedicated control and scratchpad stores normally found in microprogrammed machines to be dispensed with, thus removing a serious obstacle to microsystem support for high level languages; it also enables a far more flexible approach to be taken in capability management and program construction than has been possible in earlier systems. Whether it will be justified on balance remains to be seen.

Finally, it should be stressed that the mechanism outlined here, while appropriate to the control of program space, does not preclude the use of other abstraction devices. It would be possible, for example, to superimpose a capability mechanism extending into the file space. It would be advantageous to deal with some forms of abstraction by 'soft' methods in the confines of particular languages. On the other hand, to make the basic architecture part of a language or file system specification would be fundamentally bad design.

References

(1) *Basic Machine Principles*, J.K. Iliffe, Macdonald/American Elsevier (2nd edn 1972)

(2) *PN System Programming Manual*, J.K. Iliffe, Computer Systems Laboratory, Queen Mary College, University of London (1979)

HEIRARCHICAL FINITE STATE MACHINES AS A STRUCTURE FOR INPUT/OUTPUT SYSTEMS

HUGH L. APPLEWHITE

HONEYWELL SYSTEMS AND RESEARCH CENTER
2600 RIDGWAY PARKWAY NE
MINNEAPOLIS, MN 55413

1.0 Abstract

The input/output interface has traditionally been a source of trouble in computer systems. A heirarchical model, based on Finite State Machines and appropriate to both hardware and software, is presented which addresses these problems. This model is of interest for several reasons: first, it suggests a structure for the design of input/output subsystems; second, it is amenable to automatic manipulation using well-known algorithms (e.g. state minimization); third, it is easily and efficiently implemented in software, firmware, or hardware; fourth, automatic generation of tests is posible.

2.0 Introduction

While progress has been made in other areas of computer system design, the input/output area has been totally neglected. We speak of an architecture as being 'language directed' to indicate that it embodies the philosophy of a language. We recognize an instruction set, say, as being high level. Or we construct a memory system to ensure an abstract requirement such as security. But try as we may, no guiding principles can found for input/output systems. About the only general statement to be made is that data is transported between the outside world and the processor/memory.

High level languages have long been looked to as unifying concepts for processor and storage architecture. Significantly, input/output interfaces are programmed almost universally in an assembly language, not a high level language. It is symptomatic of the lack of progress in this area that the programs which deal with i/o are still constructed in the most primitive language. High level languages are considered to be too inefficient. This points out the lack of a unifying structure at the input/output interface.

We wish to investigate input/output interaction at the actual hardware/software interface. Previous work [1,2] has emphasized the notion of a device as an asynchronous process. This is appropriate, since synchronization is an important issue in dealing with peripheral devices. This paper, though, deals with the input/output system at a different level--the actual hardware/software interface. The two views are complementary in that we do not remove asynchronous activity from the i/o area, but rather present a more software compatible view of the i/o interface for the device processes to deal with.

3.0 Current Practice

Given that a particular piece of equipment is to be connected to a computer system, typically a hardware designer steps in and designs a controller. The hardware designer is given the device input-output characteristics; these

may involve a fairly large number of analog and/or digital lines subject to varying electrical, physical, and logical constraints. The product of the hardware designer's labors is the logical device visible to the programmer as a set of io ports or memory registers. Then a prototype is built and the hardware debugged.

Now a programmer enters the scene and designs a device driver (or handler) to connect the logical device to the operating system (and, in turn, to application level programs). The starting point for the programmer is the logical device constructed by the hardware designer. The logical device appears as a collection of bits which represent status or commands and a data register for data or addresses. The lines to the device which had a very distinct identity to the hardware designer have become a homogenous, somewhat anonymous set of bits to the programmer. In the case of the status and control bits, they may be mixed together (note that status bits are to be read, and command bits are to be written), and incidentally grouped. The problem, though, is that the programmer tends to view the device one-dimensionally. All status bits or all command bits are viewed as being equally important on the same level. But all bits are not equally important; the hardware designer understands this and, for example, will not allow the controller to function if the device is not initialized. This one-dimensional view leads the programmer to check the status of the device thru such code sequences as:

```
begin
if statusbit(1) = on   then ...
if statusbit(n) = on   then ...
    ..
if statusbit(p) = on   then ...
end ;
```

or

```
begin
if statusbit(m) = on   then ...
elseif statusbit(n) = on   then ...
    ...
elseif statusbit(p) = on then ...
end
```

or some combination of the two. In the first case, the number of possible paths thru the code for n status bits is 2**n. Note that in many device interfaces the importance of the status bits is not at all apparent - that is, there is no simple way to determine the importance of the status bits. The programmer must check bit 5, then bit 7, then bit 3, or check different sequences of bits according to whether a bit is on or not. Actually, the situation is even worse; complex devices, such as communication drivers, can require different behavior to the same status depending upon the prior history of the device.

Finally, the programmer has a driver design. He codes it and must debug it. This can be a harrowing experience, for the programmer is confronted with a new piece of hardware which may malfunction, or he may have misunderstood just how the controller works, or the hardware designer may have given him a controller which is difficult or unwieldy to deal with, or his code may be incorrect, or --- (the reader is invited to fill in other reasons). The driver may not work and he can't tell if the problem is hardware or his software. So he calls in the hardware designer to help him. But now, communication between the two may compound difficulties.

This paper will attempt to solve these problems. A finite state machine (FSM) model will be presented which is suitable for implementation in hardware, software, and firmware. The FSM has several desirable properties which make it attractive as a hardware/ software implemenation vehicle. The designer is forced to explicity account for all situations which may arise. It is sufficiently high level to serve as a common design language while hiding low level implementation details. It is amenable to automatic manipulation using well-known algoritnms [3]. Given suitable restrictions on the model (i.e., hierarchical structure), and forcing the interface registers (the logical device) to conform to a certain standard format which is particularly economical (in hardware) and efficient (in software) reduces the number of states to a manageable set. In fact, exhaustive testing may become feasible. Automatic generation of tests is also possible [4,5,6]. Lastly, the FSM collects together sufficient information to provide a history of operation which can be useful for checkout, testing, and performance evaluation.

4.0 A General Model

In designing an I/O subsystem, both data and control must be considered. At the operating system interface, control is simple and the data complex; at the device, the data is simple and the control complex. For example, an array (buffer) of words is presented to the I/O subsystem with a request for transfer. The I/O subsystem attempts the transfer and replies with either success or failure. At the lowest level, though, single words might be transferred one at a time, with an acknowledgement after each transfer. An error will cause retries or a failure status to be returned.

The general model, then, is heirarchically structured. Each level translates a single command into a set of commands to a lower level (the control consideration). Also, a data type is translated into a different, more detailed data type for the next lower level. Each level, then, successively refines both control and data to a more detailed form. Adjacent levels share common control and data structures. The next section will present a more specific model for the realization of the I/O subsystem.

5.0 Finite State Machines

The reader is assumed to be familiar with the concept of a Finite State Machine (FSM) [7]. FSMs will be briefly defined in order to present notation. We shall deal with the Mealy model of an FSM as it seems to offer technical simplifications for our purposes.

A Finite State Machine is defined as a sextuple

< S, I, O, NSF, OF, S0 >

where

S - a set of states
I - a set of inputs
O - a set of outputs
NSF - a next-state function
NSF : SxI => S
OSF - an output function
OF : SxI => O
S0 - the initial state

Where there can be no confusion, we may omit explicitly listing the various sets. We will rely on context to implicitly define them by giving the Next State and Output Functions either in tabular form as in Figure 1a or in graphical form as in Figure 1b.

An FSM operates as follows: It begins operation in its initial state. Receiving an input, it performs some output dependent on its

state and input. Then it moves to another state, again according to its current state and input. The process repeats continuously. Figure 2 shows a skeleton program which implements this process.

6.0 Heirarchical Finite State Machines

A Heirarchical Finite State Machine (HFSM) is a set of machines M[i] , i>0, such that

M[i] is an FSM :
< S[i], I[i], O[i], NSF[i], OF[i], S0[i] >
augmented by
< ES[i], ILF[i] >
where
ES is contained in S[i]
ILF[i] : ES[i] => M[j] where j>i .

As is often the case with automata-theoretic definitions, the formalism appears complex, yet the operation of the defined machine is simple.

Intuitively, an HFSM is a collection of FSMs with a mapping between the states of a machine at one level and the machines of the next lower level. That is, a state of a machine at level i may be associated (by an Inter-Level Function ILF) with a machine at level i+1. Not all states need be mapped to a lower level machine; The states that are so mapped are termed explosive (the set ES in the above definition). Figures 3,4 illustrate the structure of a simple HFSM. The HFSM operates similarly to an FSM with one exception: when an explosive state is reached, the execution of the HFSM at that level is suspended, and the submachine corresponding to the explosive state is activated. The submachine starts in its initial state, and execution commences around its state transition graph. The submachine may, in turn, contain explosive states, in which case a sub-submachine is recursively activated, and so forth. When the submachine finally makes the transition back to its initial state, the operation of the submachine ceases, and the next higher level machine (which invoked the submachine) resumes operation. Note that since the submachine initiates and terminates activity in the same state, it has no memory of previous incarnations.

We mention in passing that an HFSM is exactly equivalent to a much more complex FSM. Thus, an HFSM has no greater theoretical power than an FSM. Practically, though, it has several advantages:

1. Heirarchical structure which may be designed and implemented in a top-down fashion.

2. A clear separation of concerns (inputs-outputs) at each level.

3. An HFSM may be implemented with less memory than the equivalent FSM, since the HFSM is a collection of small FSMs rather than large FSM. The nextstate and output functions grow as the product of states and inputs, and a single FSM may require a large amount of memory to represent these functions.

6.1 Inputs

Inputs are usually specified in simple examples as single symbols, for example, '0' or '1'. Implicitly, we mean two distinct events: first, that an input is present, and second, that the input has some given value. We wish to deal with asynchronous systems, so input evaluation does not occur until an input is present.

For certain systems, a single input symbol may not be sufficient. In that case, an input can be considered to be a condition which is to be evaluated as true or false. Only one input

may be true. The single input symbol is a special case; it is simply the condition input = symbol .

6.2 HFSM Data

The previous section presented the flow of control of an HFSM. To be useful, though, it must be possible to pass data through the HFSM. Several data buffers are provided to each machine: an input-output pair to be used for communicating with the next-higher level (i.e. the invoking) machine, and an input-output pair for each explosive state to be used for communicating with submachines. Note that because the HFSM is a strictly sequential machine, one pair of data buffers may be used to communicate with all next lower level machines. Four primitives are provided for utilizing these buffers.

1. Read from Above (RA) - Read the data buffer containing data from the next higher level machine.

2. Write to Above (WA) - Write data into the data buffer of the next higher level machine.

3. Read from Below (RB) - Read the data buffer containing data written by the submachine corresponding to the last explosive state.

4. Write to Below (WB) - Write to the data buffer which can be read by the submachine corresponding to the explosive state being entered.

Note that the data passed by the Write to Below (WB) function to the next lower level, and received there by the Read from Above (RA) function, must agree in type. Similarly, the Receive from Below (RB) and Write to Above (WA) functions must agree in type.

7.0 Hardware Implementation

The implementation of an HFSM in hardware is fairly straightforward. It is similar in operation to the software version presented earlier. However, certain additions are made in order to facilitate testing and to accomodate the lower bandwidth communication channel between the device controller and main memory. Each machine may be implemented in its most convienient form--most likely as a microprogrammed controller [8]. In this conection note that the control of all machines is identical.

Each machine must provide to the software driving it the information listed in Figure XX. All fields are encoded as small integers so that simple indexed table look-ups and CASE statements may be used to access the HFSM. The machine ID field identifies the submachine. The state, input, and output fields describe the machine state and its environment. So far, the hardware implementation is exactly the same as the software version. One extra item is added to the hardware version: the machine ID interrupt level. This is a register loaded by the software driver at initialization time which specifies which machine's state transitions cause interrupts (or equivalently, when software interaction is needed).

Transitions of machines which have IDs not equal to the machine ID interrupt level proceed at their own rate. The machine specified in this register is the highest level machine in the hardware. This is the hardware macine which interacts with the software machine. The lower level machines are simple, execute quickly, and do not require software intervention.

8.0 Testing

Testing the hardware portion of the HFSM is made possible by the variable hardware/software interface, the machine ID register. In the event of a hardware failure, indicated by illegal state transitions and the like, the software can test the hardware portion of the HFSM. The test portion of the software contains a duplicate implementation of the lower levels of the hardware machines. Of course, this duplicate is not used for normal operation, but only for testing. The software sets the machine ID interrrupt level to the next lower level machine and executes a predefined test. It compares the execution of the hardware machine to its own simulation and records the differences. These differences locate the faulty state transitions. If there are no discrepancies, then it repeates the process on the next lower level machine. It continues checking lower level machines until faulty transitions are isolated.

9.0 Unresolved Issues

While HFSMs are attractive as a means of structuring hardware/software, there are several areas of conventional usage which do fit well into the model.

1. Concurrent activity cannot be expressed within the model. An FSM cannot represent concurrent threads of control. More general models, such as Petri nets, can represent concurrent activity [9] and have been used as hardware/software models [10]. But these models seem to lose some of the essential simplicity of the FSM model. Futhermore, many of the interesting properties of these models are either undecidable or computationally expensive. In contrast, FSM questions are all decidable, and for most FSMs encountered in practice, of reasonable computational expense. In passing, we note that the FSMs which we have used are fairly small (5-10 states and a like number of inputs). References [11,12] suggest a connection with synchronization using regular path expressions.

2. Certain types of exception conditions are not cleanly handled. Asynchronous exceptions arising from an external source do not fit the model well as they are not the response to some action and may occur in the middle of some conceptually indivisible action. An example of this type of condition is a power-failure indication. It is not possible to guarantee that the power fail occurs only when the machine is in certain states at some given level. One might simply include a power-fail transition in every state for every machine, but this is an unsatisfactory solution.

10.0 Conclusions

The input/output interface has traditionally been a source of trouble in computer systems. Reasons for this include a lack of communication between hardware and software designers, lack of a unifying framework for hardware and software specification, and an inability to completely test hardware/software interfaces realistically due to the large number of states involved. The problem is particularly apparent in the programs which mate a piece of hardware (for example, a peripheral controller) to an operating system.

11.0 Acknowledgements

The author would like to acknowledge Mohammed Gouda, Robert Arnold, Rick Ramseyer, and John Kellum for their time, opinions, and criticism. Denise Johnson patiently typed the author's illegible handwriting; as usual, she performed excellently under tight deadlines and conflicting demands.

12.0 References

1. Wirth,N., Modula: A Language for Modular Multiprogramming, Software Practice and Experience, Vol 7, 1977.

2. Ravn,A., Device Monitors, IEEE Trans. on Software Engineering, Vol SE-6, Jan 1980.

3. Hill,F.and Peterson,G., Introduction to Switching Theory and Logical Design, Wiley, New York, 1966.

4. Chow,T., Testing Software Design Modeled by Finite State Machines, IEEE Trans SE, Vol SE-4, No 3, May 1978.

5. Hennie,F., Fault Detecting Experiments for Sequential Circuits, Proc Fifth Annual Symposium on Switching Circuits and Logic Design, Princeton,NJ, 1964, pp 95-110.

6. Breuer,M. and Friedman,A., Diagnosis and Reliable Design of Digital Systems, Computer Science Press, 1976.

7. Gill,A., Introduction to the Theory of Finite State Machines, McGraw-Hill, New York, 1962.

8. Jensen,E. and Kain,R., The Honeywell Modular Microprogram Machine, Proc Fourth Annual Symposium on Computer Architecture, March 1977.

9. Peterson,J. , Petri Nets , Computer Surveys, Vol 9, No 3, Sept 1977.

10. Rose,C. et al, The Logos Representation System, Sixth IEEE Computer Society Conference, Sept 1972, pp 187-191.

11. Campbell,R. and Habermann,N., 'The Specification of Process Synchronization by Path Expressions', in Operating Systems, Lecture Notes in Computer Science, Vol 16, Springer-Verlag, New York, 1974.

12. Habermann,A. , Implementation of Regular Path Expressions, Carnegie-Mellon University Computer Science Department Technical Report, Pittsburg, PA , 1979.

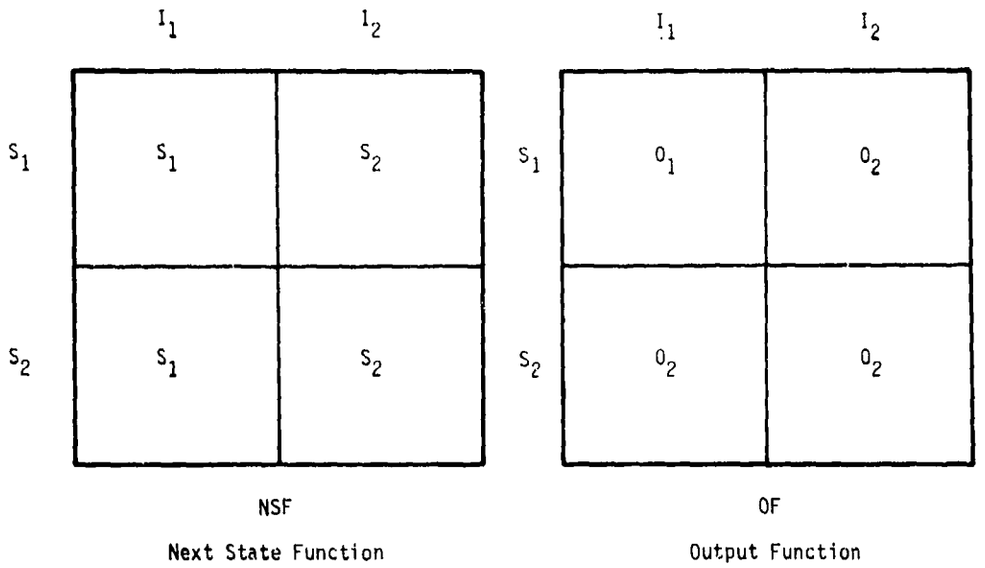


Figure 1a. Tabular FSM Description

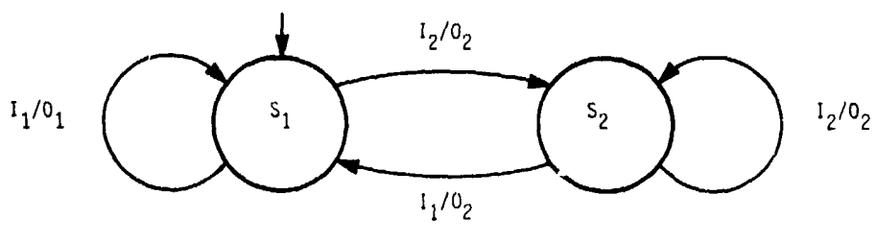


Figure 1b. Graphical FSM Description

```
procedure fsm ;
type
   nextstatetype = array[1..2,1..2] of integer ;
   outputtype    = array[1..2,1..2] of integer ;
const
   nextstate = nextstatetype (( 1, 2 )
                              ( 1, 2 ));
   output  =  outputtype (( 1, 2 )
                          ( 2, 2 ));
var
   currentstate : integer ;
   currentinput : integer ;

procedure getinput (var inp : integer);
begin { getinput }
 ...
end { getinput };

begin { fsm }
repeat
    getinput (currentinput);
    case output[curentstate, currentinput]  of
        1: ...
        2: ...
        n: ...
        end ;
    currentstate := nextstate[currentstate, currentinput];
   until forever ;
end { fsm } ;
```

Figure 2. FSM Skeleton Program.

```
procedure hfsm ;
type
   hfsms = record
        initialstate : integer ;
        currentstate,
           currentinput : integer ;
        nextstate : array[inputs,states] of integer ;
        output : array[inputs,states] of integer ;
        explosive : array[states] of boolean ;
        submachines : array[states] of integer;
        end ;
begin
currentstate := initialstate;
repeat
    getinput(currentinput);
    case output[currentstate,currentinput] of
        1 : ...
        2 : ...
        n : ...
        end ;
    currentstate := nextstate[currentinput,currentstate];
    if explosive[currentstate]
        then hfsm(submachine[currentstate]);
  until  currentstate=initialstate ;
end ;
```

Figure 3. HFSM Program Skeleton.

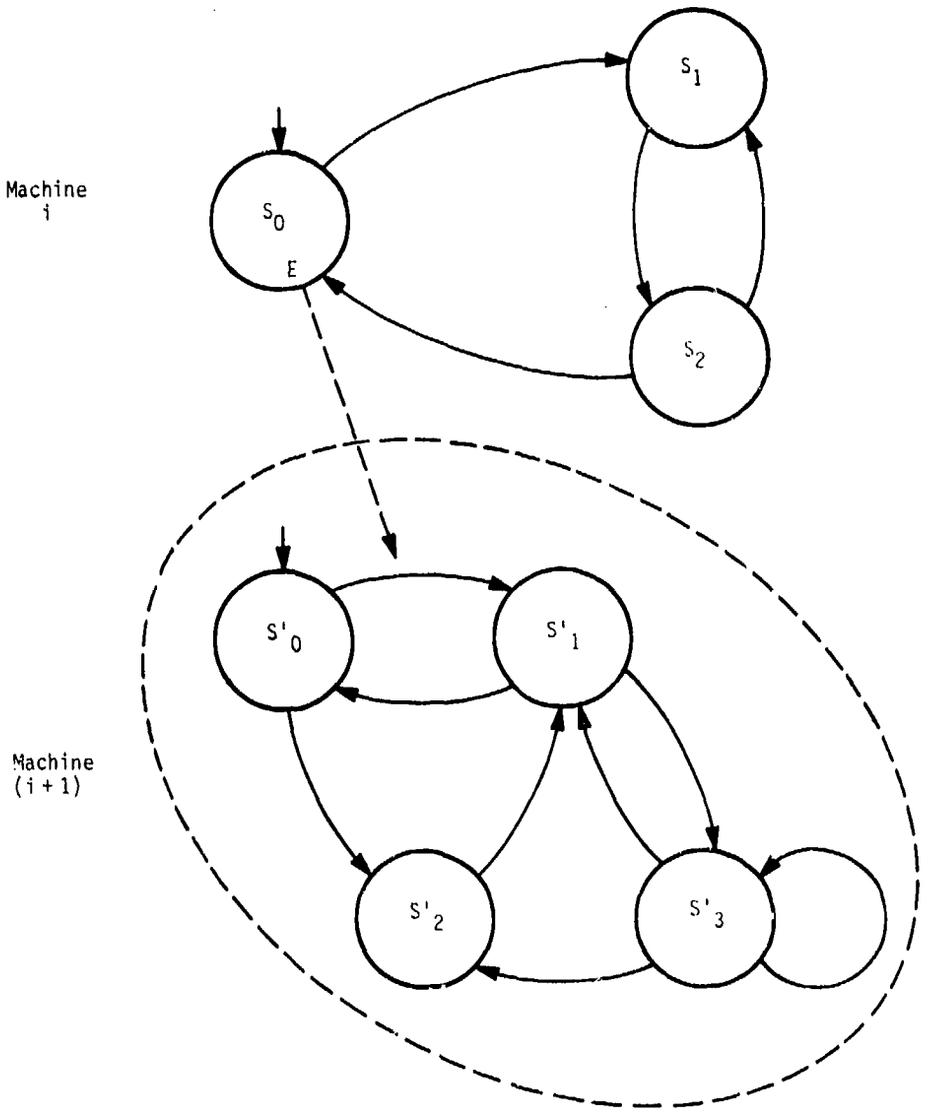


Figure 4. Simple HFSM

MARKING ID	CURRENT STATE	CURRENT INPUT	CURRENT OUTPUT

Set by Hardware

MACHINE ID INTERRUPT LEVEL

Set by Software

Figure 5. Hardware Implementation of HFSM

SWARD - A SOFTWARE-ORIENTED ARCHITECTURE

Glenford J. Myers

IBM Systems Research Institute
New York, New York

Abstract

The software problem, measured in such terms as the high cost required to develop, test, debug, and maintain programs, and the high degrees of complexity and unreliability in programs, is now the major obstacle to computing, from microprocessor applications to large-scale systems. One partial solution is bringing semiconductor technology, in the form of improved architectures, to bear on the problem. In doing so, the contention is that machine architectures should not be oriented toward just programming languages, but, more importantly, provide mechanisms on which software systems concepts can be readily based, and provide a more consistent programming environment.

SWARD, an experimental architecture, is discussed as an example of how a machine architecture can assist in the solution of the software problem.

Introduction

There is widespread agreement that the development of software is the largest problem in the computer field today. The problem is manifested in the following ways. First, the production of software is a costly venture. The great leaps forward in the cost of digital hardware have not been experienced in software development. Where, in the past, the software cost of a computing system was outweighed by hardware costs, the opposite is the case today. For instance, the cost of producing a single instruction in a program for a microprocessor system probably exceeds the cost of the processor.

Second, in typical software-development projects, more than 50% of the development costs are expended in the testing and debugging processes. Furthermore, the maintenance costs of a production program often exceed its development costs.

Third, error rates in the software design and coding processes of one mistake per 20 statements, and worse, have been reported in the literature. Hence, a program of significant size, such as 100,000 statements, might initially contain 5000 errors prior to inspections and testing.

Finally, because of the increasing sophistication of computer applications, software errors can have rather serious consequences.

These problems will be exacerbated in the future by the increasing sophistication of new computer applications in such areas as artificial intelligence, defense systems, transportation and energy management, and electronic fund transfer.

Software engineers and computer scientists have been wrestling with the software problem for the last decade. Although improvements have been made in some environments and organizations, the problem is still a serious one. One reason is the recent explosion of the amount and types of programs being produced. Ten years ago, the typical programmer could be found producing a simple Cobol application or developing an operating system for a computer manufacturer. Today we find a much larger programmer population developing such applications as chess-playing programs for consumer games, fuel/air mixture regulators in automotive microprocessors, coronary-analysis programs in medical equipment, collision-avoidance algorithms in aircraft systems, guidance programs in nuclear missile warheads, and dispatching systems for police and fire equipment. Another reason is that the largest areas of software-engineering research, namely improvements in programming languages and mathematical proofs of program correctness, have not yet had a significant effect on the software-development process in industry.

Given this situation, it seems time to exploit semiconductor technology to assist in the solution of the problem. There is ample motivation for the hardware

designer to be interested in doing so. Given the continuing reduction in hardware costs, the processor manufacturer must sell its product in increasingly larger volumes. Doing so requires increasingly larger amounts of software, and requires movement of computer technology into new application areas. The rate of sale of computer hardware, from microprocessors to large-scale systems, is directly related to how quickly the required system and application software support can be produced, and the reliability of that software.

An Approach to the Problem

The answer to how hardware technology might help alleviate the software problem is not the simplistic approach of "moving software to silicon," since there is no evidence that the problems mentioned above will disappear by merely shifting responsibility for the design task from the programmer to the circuit or logic designer. Rather, the answer is designing machines that provide less-hostile environments for programs, programmers, and end users. The architect must now face up to broader considerations, such as

1. Ways in which the architecture can simplify the task of application programming, for instance, by providing support for more-potent concepts of input/output and data manipulation in programming languages.

2. Ways in which the architecture can encourage the use of good software design and programming practices, for instance by providing efficient support for concepts of program modularity, information hiding, abstract data types, and structured programming. The motivation here and in point 1 is the prevention of programming errors.

3. Ways in which the architecture can assist the costly processes of software testing and debugging, for instance by detecting or preventing common programming errors and by providing a more-flexible base for the development of software testing and debugging tools.

4. Ways in which the architecture can reduce the complexity of one of the most-complex classes of software, namely compilers. Such support involves reducing the semantic gap bewteen languages and the architecture by tailoring the operations and objects provided in the architecture more closely to the corresponding concepts in programming languages.[1]

5. Ways in which the architecture can reduce the complexity of another complex class of programs - operating systems. This might imply increased awareness in the architecture of such concepts as protection, process management, process synchronization and communication, and memory management.

Considerations such as these have been addressed in the literature [illegible] have had little impact, as yet, on most commercially available computer systems.

The SWARD Architecture

An example of an approach to solving the software problem is an experimental system under development at the IBM Systems Research Institute. Although the current definition of the architecture has not been published, it has evolved from earlier published versions.[1,2,3]

The five sets of considerations listed in the previous section are the design objectives of the architecture. Detailed objectives were derived for each of the categories. Many of these objectives are mentioned in the following discussion of the architecture.

The major attributes of the architecture, and some of their relationships to the software problem, are outlined below.

Tagged storage. The concept of tagged, or self-identifying, storage is used throughout the architecture to allow the machine to understand unambiguously the attributes of the operands of an instruction. This allows the machine to detect operations on incompatible operands and to perform automatic data conversions during instruction processing. Each data type has a unique representation for the "undefined" state, allowing the machine to detect attempts to use undefined values.

The tagged data elements (called cells) are variable in size. The architecture contains no fixed-size word concept and permits machine instructions to address only cells as operands; hence the data model provided by the architecture closely corresponds to the data models in programming languages.

Nested tags. The tagged storage concept was extended to allow tags to be embedded within other tags, allowing the representation of higher-order data types as arrays, structures/records, and user-defined types. The machine, rather than the program, handles the task of array addressing, and automatically performs bounds checks. The architecture also contains explicit representations of

arrays of structures/records and "based variables."

Capability based addressing. The architecture employs the addressing and protection concept of capability-based addressing. The architecture views the world as a set of objects, each being given a unique name by the machine when created. Programs cannot fabricate or manipulate addresses, and any reference to an object after the object has been destroyed results in a detected error.

Capabilities and objects are used to create a high-level storage model, the elimination of traditional low-level storage concepts being another objective of the architecture. Figure 1 depicts a possible state of the storage model. The architecture recognizes five types of objects, four of which (module, process machine, port, data-storage object) are explicitly created and addressed by programs and one of which (activation record) is implicitly created via a module invocation.

Full generality of allowing capabilities to reside in objects is provided; capabilities are protected by their being one of the 15 tagged cell (data) types. As shown, the architecture also uses capabilities to reference source/sink (storageless) I/O devices.

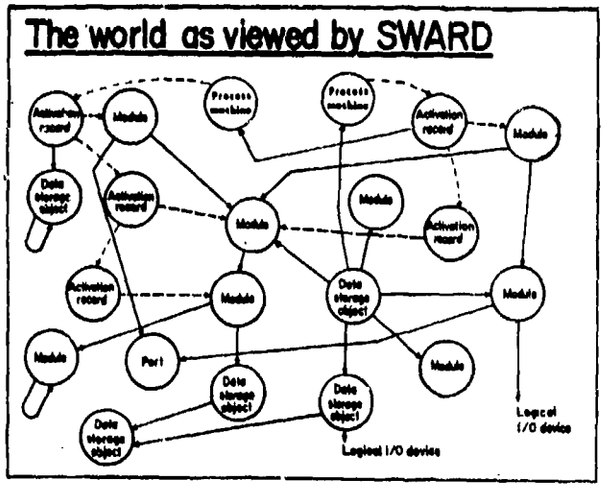


Figure 1

Single level storage. The concept of virtual storage has been generalized to the extent that there is no notion, above the architecture, of secondary storage. For instance, the concept of files no longer exists; programs use arrays to represent what would have been considered to be a file. Hence the concept of secondary-storage I/O has been eliminated; all data in the system are addressed in a uniform way, and all other concepts in the architecture (e.g., tagged storage) apply to all data in a uniform manner.

Within the environment, all concepts of storage allocation have been removed from the domain of software. Although storage allocation does occur, it is done implicitly by the machine, for instance, as an effect of a module invocation (where the machine creates an activation record for the module's local variables). Rather than being able to allocate space, programs are presented with a function to allocate occurrences of cell types, such as strings and arrays (the dynamic allocation of which is embodied in a data-storage object).

Small protection domains. Each subroutine or procedure of a program is represented by a module object, which contains the generated instruction stream and a definition of the module's address space (a set of tagged cells). This structure is shown in Figure 2. Instructions in a module can address items only within the private address space, although well-controlled indirect references can be made, via parameters and capabilities, outside of the address space. Thus the architecture enforces rules of program modularity, limits the consequences of errors, and protects a program, including the system software, from itself.

Automatic subroutine management. The architecture removes the burden of subroutime management from the shoulders of the compilers by containing instructions that perform all that is implied by a subroutine call in a high-level language. For instance, the CALL instruction saves the state of the current module, creates and initializes an activation record for the called module, switches address spaces, and begins execution of the called module. The attributes of arguments and parameters are verified for consistency during each call.

Figure 2 shows that a module's address space is partitioned into two sections - the "static storage die" and "automatic storage die." Cells in the static storage die reside permanently within the module object. When a module entry point is called, the machine creates

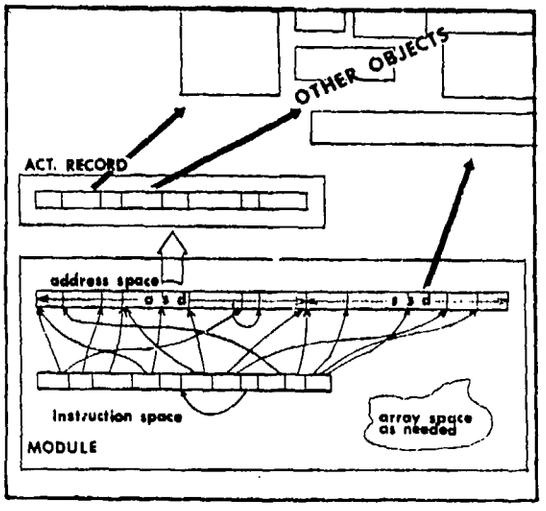


Figure 2

an activation-record object containing a copy of the definition of the cells in the automatic-storage-die part of the address space. When an instruction refers to a cell in the automatic storage die, the machine automatically maps this reference to the corresponding cell in the current activation record.

Hierarchical fault-handling mechanism. The architecture contains a uniform, process-oriented, rather than system-oriented, mechanism for the handling of error conditions, called faults. Any module can contain a special fault-handling entry point and specify which types of faults can be handled there. When a fault is detected in a module, the machine searches back through the activation history of the process, looking for the first module that has indicated a desire to handle that type of fault. When one is found, the machine "calls" that entry point (i.e., simulates a subprogram call), passing it five arguments describing the fault and the state of the program at the time of the fault. What happens after that is a function of the fault-handling software in the module. However, the architecture provides several instructions to terminate a fault handler and an instruction to explicitly raise fault conditions.

Process machines. One of the five types of objects defined by the architecture is a process machine, an abstract entity that executes a concurrent process. A process-machine object has the characteristics of a hardware processor and thus creates a multiprocessor environment; however, the mapping of process machines to hardware processors is a matter of hardware implementation, not architecture. (At one extreme, a single hardware processor can time-slice itself to act as all process machines.)

By creating and destroying process machines, programs create and destroy processes. In keeping with the design rules followed throughout the architecture, this entity defines only a mechanism, out of which programs can create policies. Also, it is orthogonal with other concepts in the architecture (e.g., process machines have no relationship to addressing).

Send/receive mechanism. Two machine instructions, SEND and RECEIVE, and an abstract object, a port, are provided for interprocess communication. The SEND instruction is defined almost identically to the CALL instruction, except where CALL transfers control and a set of arguments to a module entry point, SEND transfers a set of argument values through a port. That is, it transfers data but not control. As with the subroutine call mechanism, type checking occurs across the send/receive interface. As mentioned earlier, source/sink devices are represented by capabilities, and one does I/O operations on these devices by use of SEND and RECEIVE.

The mechanism is synchronous to the extent that a process machine executing a SEND instruction halts until another process machine receives the transmitted values. Thus the mechanism is similar to the rendezvous concept in the Ada language.

Generic instructions. The concept of tagged storage allows the architecture to be defined with a small, highly regular, generic instruction set. For instance, there is only a single instruction for performing addition - ADD - and only a single instruction for transferring values in storage - MOVE. The semantics of the instructions are defined by the attributes of their operands. For instance, the MOVE instruction can be used to store an integer value in a floating-point data cell (doing an automatic data conversion), store one character string in another, store a scalar value into all elements of an array, or set one array equal to another. One of the benefits of this is significant simplification of compilers, particularly the code-generation process.

Powerful instruction repertoire. In addition to the instructions mentioned earlier, the architecture contains an

instruction to address and move substrings within strings, a search instruction to search an array for a matching value, and an iterate instruction embodying the full semantics of iterative DO loops in such languages as Fortran and PL/I.

For process synchronization, the architecture contains two instructions named GUARD and UNGUARD. They can be used to prevent simultaneous execution of two or more processes through a critical section of instructions and were motivated by the software design and synchronization concept of monitors.[12]

Transparent indirect addressing. The concept of capabilities has been expanded to allow capabilities to point to other capabilities such that, if a program refers to a capability, the machine will interpret this as a reference to the last capability in the chain. This concept can be used for added levels of data security, by an operating system for access control of objects, and to allow one to dynamically replace objects (e.g., modules) in a program while the program is executing.

Program tracing facilities. Instructions exist to activate the tracing of branches taken, branches not taken, and/or calls in specified modules. When such events occur, they are treated by the machine as faults and thus the fault-handling mechanism mentioned above applies.

Additional security features. In addition to the protection concepts of capabilities, small protection domains, and indirect capabilities, the architecture contains additional security features, such as the ability of a program to restrict the copying of capabilities, an instruction to assign a new unique name to an object, and a second level of protection provided by the use of tagged storage.

Semantic checking. One of the major objectives of the architecture is detection of large classes of semantic errors in programs, errors that are (1) frequent, (2) difficult to debug when they occur in conventional systems, (3) common to many or all programming languages, and (4) in general, not detectable at the time of program compilation. Examples of a few of the 27 classes detected are (a) use of undefined data values, (b) references to nonexistent array elements, (c) the dangling-reference problem, (d) data type ambiguities (e.g., inconsistent declarations of global data), and (e) mismatching arguments and parameters. Studies have indicated that these errors represent 30-50% of all errors in typical programs.

Virtual machine. Although not an explicit objective of the architecture, attributes of the architecture, such as capabilities and objects, have given it the characteristic of being a virtual-machine environment, meaning that programs can exist having no relationship to the operating system, and multiple operating-system environments can coexist.

Relevance of SWARD to the Software Problem

The SWARD architecture is unique in that almost every aspect of the architecture was motivated by a desire to alleviate the software problem. The major ways in which this is achieved are discussed below.

The extensive semantic checking performed by the machine should enhance significantly the productivity of the software testing and debugging processes, and lessen the consequences of errors occurring in production programs.

The object orientation of the architecture, and the use of capability-based addressing, presents a highly uniform system environment. The objects of the architecture (modules, process machines, ports, data-storage objects), as well as source/sink I/O devices, are addressed in an identical fashion. This has important implications on the complexity of system software and the user environment. For instance, where conventional systems contain a variety of dissimilar mechanisms for the binding of entities (e.g., a "linkage editor" for binding program modules together, control-language statements and "open" services for binding programs to files), an operating system can be defined with a single uniform concept of binding.

The single-level store concept, particularly when carried forth into programming languages, largely eliminates the need for I/O concepts, allowing the programmer to think of data in a uniform way.

The use of the SEND and RECEIVE instructions as the basic I/O primitives for source/sink devices, as well as for interprocess communication, has several benefits. First, it adds another measure of uniformity to the system, since, for instance, there is no difference among sending a character string to a printer, terminal, or another process through a port. Hence there is only one concept of data transmission. Second, it allows one to substitute processes for I/O devices,

or I/O devices for processes, without changing one's program. Third, the send/receive mechanism is synchronous with respect to whatever is on the other side (I/O device or process). Hence there is only one concept of parallelism in the system - the process. There is no concept of an interrupt.

Other unifying ideas, all of which serve to make the programming environment a less-complex and less-hostile one, are the fault-handling mechanism for error handling, capability-based addressing for information sharing and protection, the highly generic instruction set, and no need for a privileged instruction state.

The development of well-structured programs, employing concepts of modularity, information hiding, and parallel processes, is encouraged by the machine concepts of an efficient subroutine-management mechanism, small protection domains, the fault-handling mechanism, the single-level store, the GUARD, UNGUARD, SEND, and RECEIVE instructions, and others.

The points above apply to the programming environment in general, but several additional points can be made about compilers, operating systems, and data-base management. Because of the concepts of tagged storage, direct recognition of higher-order data types such as arrays and structures, the generic instruction set, and the power of the instruction repertoire, the development cost and complexity of compilers should be significantly reduced.

For many of the same reasons, and because of other facilities in the machine, the overhead and development cost of high-level-language-oriented testing and debugging tools should be greatly reduced.

The architecture also eliminates much of the traditional complexity of operating systems and other subsystems by removing from them the problems of memory management, protection, process synchronization, interprocess communication, and interrupts.

The use of generic instructions and tagged storage implies the latest-possible binding of instructions and data; the semantics of an instruction are determined at the time of its execution, using the information in the tags of its operand cells. SWARD extends this even further by allowing the programmer to incompletely specify the attributes of a local variable in its tag; this allows a local variable to acquire dynamically some or all of its attributes (e.g., from a parameter). These points have significance to the concept of data independence in data base environments.

Conclusion

Given the magnitude of the software problem today and an appreciation for how much worse it will be tomorrow, and given the rapid advances in hardware technology, the time seems ripe for major architecture redirections that make fundamental improvements in the programming environment. The SWARD architecture serves as an example of how a machine architecture can reduce software complexity and lessen the difficulty and error-proneness of program design, coding, testing, and debugging.

References

1. G. J. Myers, Advances in Computer Architecture. New York: Wiley, 1978.
2. P. J. Denning, "A Question of Semantics," Computer Architecture News, 6(8), 16-18 (1978).
3. E. A. Feustel, "On the Advantages of Tagged Architecture," IEEE Trans. on Computers, C-22(7), 644-656 (1973).
4. J. R. Ehrman, "System Design, Machine Architecture, and Debugging," SIGPLAN Notices, 7(8), 8-23 (1972).
5. P. Brinch Hansen, "Multiprocessor Architectures for Concurrent Programs," Computer Architecture News, 7(4), 4-23 (1978).
6. K. Berkling, "Computer Architecture for Correct Programming," Proc. Fifth Annual Symp. on Computer Architecture. New York: ACM, 1978, pp. 78-84.
7. H. J. Saul and L. J. Shustek, "On Measuring Computer Systems by Microprogramming," Microprogramming and Systems Architecture: Infotech State of the Art Report 23. Berkshire, England: Infotech, 1975, pp. 473-489.
8. J. L. Keedy, "A Technique for Passing Reference Parameters in an Information-Hiding Architecture," Computer Architecture News, 7(9), 11-15 (1979).
9. G. J. Myers, "The Design of Computer Architectures to Enhance Software Reliability," Ph.D. dissertation, Polytechnic Institute of New York, 1977.
10. G. J. Myers, "Storage Concepts in a Software-Reliability-Directed Computer Architecture," Proc. Fifth Annual Symp. on Computer Architecture. New York: ACM, 1978, pp. 78-84.
11. J. B. Dennis, "Computer Architecture and the Cost of Software," Computer Architecture News, 5(1), 17-21 (1976).
12. C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," Comm. of the ACM, 17(10), 549-557 (1974).

CONSIDERATIONS IN OPERATING SYSTEM DESIGN FOR MULTIPROCESSOR STRUCTURES

HAROLD LORIN — IBM SRI

BARRY C. GOLDSTEIN — IBM Research Center

ABSTRACT

Given the advances of technology, it is not unreasonable to project the existence of multiple processor configurations that have large numbers of processors with a variety of interconnection possibilities.

This paper discusses language constructs for interprocess communication and process creation functions which would be fundamental to systems that run sets of programs dispersed across families of logical processors. Certain divergences between various concepts of interprocess communication are resolved in a single design.

INTRODUCTION

Recent dramatic developments in processor/memory technology and in interconnection methodologies for the association of processors with each other suggest that future multiple processor configurations may have *large* numbers of fast and cheap processors with a variety of memory sharing possibilities[1,2,3,8].

An objective of such multiple processor systems will be the need to quickly and dynamically react to the changing demands on the system. This will imply the need to not only group a set of processors to work on a given set of applications but will also imply the need to dynamically partition memory spaces which are physically common amongst these set of processors. For convention we will refer to a set of processors and memory formed dynamically as a *logical system*.

In such systems sub-configurations of closely cooperating generic multiprocessors may be formed and partitioned sets of 'distributed' configurations may be formed between units with a rich diversity of decisions about memory sharing, code replication, etc. The intent of the concept, of course, is to allow systems to take shapes appropriate for their applications. To support this notion various speeds of memory and processors would be available so that various concepts of application speed and partitioning can be supported by decisions about processor and memory speed and capacity.

Some important concepts of dynamic configurability should exist in the system. Sets of closely cooperating, memory-shared processors should be dynamically definable for short-periods, cooperating or independent sub-configurations of "distributed" systems should also be definable for brief periods. Where desirable, permanent "gangs" of associated processors at different levels of memory and operating system sharing should also be definable within the total population of processors, memories and other resources of the system. A goal of such a system is to make maximum use of the well known concept that logical systems structures of varying kinds of relationships and closeness of cooperation can be mapped onto physical structures.

DESIGN CONCEPTS

A very well known way of structuring an operating system is to define vertical partitions of functions such that there is a functional module for I/O, memory management, process communications, process synchronization, etc. The great advantage to the structure, of course, is that it allows multiple parallel services to be achieved in multiple processor environments.

The structure can be supported by hardware in a number of ways. Each functional module can be located in protected address spaces in a large single physical processor/memory. An interesting attribute of a capability, object management architecture such as SWARD[6] is that the physical configuration of memory is logically irrelevant. Configurations can be formed with various degrees of shared or private physical memory without impacting the logic of the object management system.

The ability to assign some number of processors of any architecture to a system suggests that these processors may be used as Global Service processors, each assigned to a significant operating system function of the type suggested above. There may be a Systems Wide Message Handler, a Systems Wide Global Scheduler, A Systems Wide I/O server, etc.

In an alternative structure, each processing node could be composed of two processing nodes. Conceptually one might think of a Problem State element and a Supervisor State element. All those activities which would be executed in supervisor state in S/370 architecture would be executed in one element, while all those in problem state in another element. Although this serves as a conceptual example, it is not clear that this particular partitioning of function between elements of a node is the proper partitioning point. The discovery of a proper partitioning between computational element and operating system element depends upon a number of factors which include frequency of function, instruction set restrictions, the degree of asynchroniety, etc. A fall back concept is to view the operating system element as a kind of network processor which becomes involved only when the associated computational processor issues a request which will involve interaction with another station in the network. This may be falling back too far since it places in the computational processor the burden of determining when an off-station reference must be made and this effort may be large compared to making the interaction itself. It is preferable for the operating system processor to determine what and when off-station references must be made while the computational processor proceeds with other available work.

In such a system, where each node is comprised of at least an operating system processor and a computational processor, each operating system element has a functionally equivalent local operating system that participates in global system decisions and global system services as well as providing local support. This constitutes the basis for a completely distributed control system in which intensive interaction is sustained between stations and where negotiation and co-operation lead to system wide decisions about work distribution. A best processor for a unit of work may be discovered by interaction and negotiation with operating system elements aware of what their associated computational processor is doing. This negotiation goes on *without* disturbing the progress of available work on a local dispatch list. The following sections address these design objectives with respect to the language constructs and control structures for process creation and inter-process communication.

TERMINOLOGY

In the system we are about to describe we introduce the following terminology:

PROCESS CONTROL TERMINOLOGY

APPLICATION - An *application* defines a context by indicating a set of procedure and data objects that may be accessed and states the rules of reference. The *application* describes both the physical and abstract resource constraints necessary and permissible for processes belonging to the *application*.

PROCEDURE - A *procedure* is program text and capabilities for an incarnation of a process or an instance of activation. Its unique feature in this system is a statement of consumable resource constraints in addition to the list of abstract resources (such as files, data bases, locks, etc) necessary for successful processing.

PROCESS - The System dispatchable unit. A stack of activation records, each associated with a procedure, resting upon a *process* activation block that may be used for recovery. A *process* is named.

USER - Each *user* of the system is, of course, defined to the system. Part of this definition is a list of the total set of *applications* which the *user* can connect to.

PROCESS COMMUNICATION TERMINOLOGY

PORT - A *port* may be a *to_port* or a *from_port*. On a sender's side a *from_port* is a named place in a sender's program (e.g. a declared structure in the PL/I sense) serving as the source of a message to be sent. A *to_port* is the name of a receiving *process*. On a receiver's side a *to_port* is a named place in the receiver's program where the message will be placed. A *from_port* is the name of a sending *process*.

PATH - A *path* can be either a queue name or a file name. The *path* represents an indirection from the sender to either a specific receiver or to an arbitrary receiver.

The exact details of inter-process communication will be deferred till the section on process communication.

PROCESS CREATION

One of the major objectives in introducing new language constructs is to insure that in so far as is reasonable the language for application programming is the same as the user's command language. Not having this as an objective results in increased complexity in requiring a user to learn more than one language for performing the same identical function. For simplicity, PASCAL is used as the language for syntax expression in this section and in the next section [7,9], though the language constructs presented are not unique to PASCAL.

In the system we are presenting, there are users who initiate processes (which can initiate still more) which run under a given application scope. Consequently, the following declarative structures:

```
type user = record
                application_set: set of application;
                default_appl: application;
            end

type application = record
                name_space: set of name_pair;
                default_process: procedure;
                processor_resource: proc_req;
                abstract_resource: set of resource;
            end

type name_pair = record
                name: alfa;
                object: object_descriptor;
            end

type proc_req = record
                min_processors: integer;
                max_processors: integer;
                min_memory: integer;
                max_memory: integer;
                instruction_set: machine_type;
                performance: set of perform_req;
            end

type procedure = record
                entry_point: program;
                name_space: set of name_pair;
                processor_resource: proc_req;
                abstract_resource: set of resource;
            end
```

The fields in the above records are described as follows:

USER The *application_set* is a list of *application* names. These represent the total set of allowable *applications* that a given *user* is allowed to access. The *default_appl* is the *application* that a *user* will be automatically connected to when he *LOGONs* to the system. This field is optional. Creation of an object of type *user* assuming the creator has the 'right' to create such an object, (e.g. var *hal, harry: user*) results in the system creation of a *user* object. When a *user* issues a *LOGON* to the system (e.g. *LOGON hal*), the system searches for the *user* object named *hal*. If not found then the *LOGON* is rejected, otherwise the *user* object is searched for a *default_appl* name (e.g. *hal.default_appl = null?*). If specified, then the *user* will be connected to an instance of that *application* (one will be created if it does not already exist).

APPLICATION - An *application* defines the universe of accessibility for all processes and users connected to it. The *name_space* is therefore, a set of *name_pairs* (representing the objects that can be accessed). The first element in the pair is the *name* of the object, and the second is the descriptor of the object mapped to by the *name*. Included in the descriptor are the rights of access (such as the primitives Read, Write, Execute), and the type of the object (such as queue, procedure, file, nested name space, etc).

The *default_process* is the name of the *procedure* to be invoked (as a process) when an instance of the *application* is created. This field is optional and allows an *application* to implicitly initialize its relevant structures. Of course, resolution of the name is through the defined *application.name_space*.

The *processor_resource* represents the processor requirements of the *application* and is self explanatory. Upon completion of the creation of an instance of an *application*, processor resources are allocated to the *application*. The allocated set is referred to as a *gang*. All created processes that belong to an *application* run in the *gang* associated with that *application*.

The *procedure* represents a model for either the creation of a process (either implicitly or through usage of the *START* command to be discussed below) or the creation of an activation within a process (through the standard program call interface). The processor requirement specification in the *procedure* allows for the tailoring of a given process to the consumable resource requirements of the program. Specifically performance objectives such as priority, degree of I/O boundedness, deadlines, etc can be stated in the performance requirement of the *procedure*. The *name_space* defines the scope of accessibility of the activation spawned from the *procedure*. If a *name_space* is not present in the definition then the caller's *name_space* is assumed. It should be noted that there need not be an intersection between *application.name_space* and a *name_space* defined in a *procedure* whose *name* is in *application.name_space*.

Procedure.abstract_resource identifies which resources, such as data bases, files, locks, etc have to be allocated before process/activation creation can occur.

Given this basis, we can now address process and *application* creation. Previously, we have shown how an *application* (and its default process) can be created as a result of a *user* performing a *LOGON*. This in itself is not novel and is typical of many interactive systems.

We we will now discuss explicit process and *application* instance creation. The command *START* is used for both process and *application* instance creation, and has the following format:

START variable, name

This command is identical to the form that would be used within a process to create another process or *application*. The variable is the name of the entity being started. If the issuer is not already constrained within an *application* instance then *START* searches the *user* block to determine if the variable is a valid *application* name. If it is not then the request is rejected. Otherwise, an *application* instance is created and resources allocated (note: the consumable resources allocated are transparent to the caller and are only known by the system). If the *application* has a *default_process* defined then that process is implicitly created.

If the issuer is already constrained to a given *application* instance (either through a *LOGON* or *START*) then the variable is initially treated as a *procedure* name. In this case, a search is made from the *application.name_space* (for a first time process creation) or from the *name_space* associated with the issuing process. If a *procedure* is not found from the search, then a search is made from the *user* block treating the variable as an *application* name. If the *procedure* is found then a process is created. The caller is not aware of where the created process is running.

The *name* specified on *START* is the caller known name of the created entity. If a process was created then the *application.name* represents the unique name of the created process. The *START* request will fail if the caller specified *name* is already associated with another process in the *application* instance. The usage of this name will become apparent in the discussion on inter-process communication.

What has been shown is a very simple way to effect process creation. *Applications* and processes can be created and as a result logical systems can be formed dynamically and without explicit installation intervention. The decision over whether the logical systems are distributed or tightly coupled becomes purely one of *application* and *procedure* definition which, of course, can also be dynamically modified.

Dynamic changes to resource consumption rights and processor scheduling constraints associated with any creation of a *procedure* may be made by simple use of the declarative structures of the language.

The scheduling constraints which may be associated with *START* suggest that rather complex global systems management of the type associated with large scale uniprocessors may be a feature of a multiple processor aggregative system. These scheduling rules may be enforced by a global systems scheduler node or by cooperative interaction between a set of operating systems processors which are associated with the computational processors of the system on a one-to-one or one-to-many basis.

INTER-PROCESS COMMUNICATION

Given processes the next step is in providing them with a means for effecting inter-process communication. For this function we will postulate the existence of an Inter-Process Communicator (IPC). The IPC can exist either as central service processor or can exist as a distributed service in each of the logical systems defined. Its physical existence is literally irrelevant. What is important is that the services it provides remains invariant no matter where the IPC physically resides. Thus no application recoding should be required if, for example, the decision to have a global IPC proved to be wrong.

We will postulate a Send/Receive mechanism with five verbs: *CONNECT, SEND, RECEIVE, SIGNAL, and DISCONNECT*. The concept will include the idea that processes may communicate with each other directly, or through named objects in a synchronized or asynchronous manner. The ability to send and receive between processes and objects permits I/O to be subsumed into the communication mechanisms. Connect establishes a path between an issuing process and any named object of the system. Thus a process may subsequently *SEND* messages to another process or a named data object. Messages sent to other processes may be passed through queues or sent directly to ports of a receiving process. A Receiver may ask for messages from a data object, a queue, or another process. One to many relationships may be defined to support message broadcasting, handling of a message from any of a set of possible servers, etc.

An important potential feature of an inter-processor communication (IPC) mechanism on an architecture with self-describing data [6] is the use of the *CONNECT* verb to describe a message template which provides a description of the message structures which are to move between particular *ports*. A common problem in IPC mechanisms is uncertainty about whether a sender or receiver is at fault when message formats do not match. This may occur because of programming errors which cause a wrong *port* or queue for transmission or receipt of a particular message. The provision of a message template gives the IPC a means of determining whether wrong messages or badly formed messages are the responsibility of the sender or receiver. In systems where data is self describing, an IPC can check the tags of a message for conformity with the message template. Each receiver transmits a template of

the expected message format which is also checked against the message template provided by the *CONNECT*, which has the following format:

CONNECT connect__point

where:

```
type connect__point = record
                        ports: set of message__areas;
                        path: set of (queue,file);
                        message__template: message__format;
                        case (send,receive) of
                          send:(to__port:process__name);
                          receive:(receive__point:procedure;
                                   from__port:process__name);
                        end
                      end
```

The *CONNECT* verb can be used by both Senders and Receivers (hence the usage of case in defining the *connect__point* type). *Ports* specifies the location of the message areas in the issuing process to be used either as the location for receipt of messages or for the submission of messages.

Path is optional and specifies an indirection point in the transmission of the message. The object of the *path* is physically owned and managed by the IPC. Usage of a *path* in the transmission of a message guarantees the recovery of that message. Queues are transient and exist as long as the process which requested its creation (this process can be different from either sender or receiver and could represent a caretaker process). Files are permanent and have to be explicitly destroyed. Submission of a message through a *path* guarantees, in general, the persistence of that message even though the sender and potential receiver go through untimely termination. Both FIFO and LIFO queueing techniques are applicable with queues and files and is specified when the object is created.

The *message__template* specifies the sender/receiver's model for the message. It describes, for example, the length of the message, what the encoding of the message is (ASCII, Fixed Decimal, Packed, etc), and its format (for a multi-segmented message). This template is used by *CONNECT* for comparison with a template associated with the *path*. A sender's and receivor's template is compared with the *path* template. If there is a disagreement between the *path* template and that of sender or receiver, the *process* with the divergent template is notified of a message type error. If there is no *path*, the sender's and receiver's templates are compared with each other. In case of an error both processes are informed of a mismatch. This feature is most practical for hardware systems that have strong features of self-describing data and tagged memory.

If the case is for *SEND* then *to__port* refers to the process name of the process that is to receive the request.

If the case is for *RECEIVE* then the *receive__point* refers to a *procedure* that is to be invoked when another process issues a *SEND* (not through a *path*) to that process. The *receive__point* represents a point of interruption for asynchronous receipt of messages.

It is possible to support IPC without a *CONNECT*. If no *CONNECT* is issued, processes may communicate directly or indirectly using the usual capability control mechanisms of the operating system which provide for acquiring names of processes and path objects. In this usage, full specification must occur with *SENDs* and *RECEIVEs*. The penalty for such use is increased risk of run time failure.

If *CONNECT* is used the parameters may come from:

1. totally from the sender
2. totally from the receiver
3. in some combination of both

In case (1) or (2) the parameters associated with the *CONNECT* are imposed by the system upon the relationship. Case (3) raises interesting considerations that have not yet been fully explored, as to the degrees of freedom between *SEND* and *RECEIVE* parameters. For example, a *CONNECT* issued by a receiver that names a path could be considered inconsistent with a *CONNECT* issued by a sender which did not name a path. However we may convince ourselves that there is some advantage in having transparent to one side of the send/receive relation.

The operation of sending a message can now be described:

SEND token, from__port, path, to__port

SEND has four operands. The *from__port* specifies which message areas, in the sending process contains the transmission message. The *path* specifies an indirection path for the message (as described above) and the last operand, a *to__port*, identifies the process that will receive the message. *SEND* automatically *blocks* the issuer until either the message has been placed on a *path* (if specified) or the receiving process (if no *path* has been specified) has received the message.

Specification of the three operands *(from__port,path,to__port)* are optional and can be derived from the preceding *CONNECT*. Their inclusion on *SEND* is to allow an area to be used to send messages to more than one *connect__point*.

In fact, if *to__port* is not specified in either *CONNECT* or *SEND*, then *path* must be specified in either. In this case, the message will be placed on the queue or file by the IPC and the sender will be *SIGNALed* to remove it from the *blocked* state. Such messages can be removed by any process which has IPC access to the *path*.

If *path* is not specified in either *CONNECT* or *SEND*, then a *to__port* must be specified. In such a case, the message is sent directly to the receiving process (if it has an outstanding *CONNECT* or *RECEIVE*). If there is no outstanding *RECEIVE*, then the *receive__point* identifies the *procedure* to be invoked and an activation is immediately created and made the current one. The deblocking of the Sender is then the responsibility of the *receive__point* activation which should issue a *SIGNAL* to indicate receipt of the message. If a *RECEIVE* has not been issued and the *CONNECT* does not define a *receive__point* then the IPC will implicitly queue the message, leaving the sender *blocked*, until a *RECEIVE* is issued. It is still the receiver's responsibility to deblock the sender. The systems events that occur when there is an outstanding *RECEIVE* are discussed below when we describe *RECEIVE*.

The *token* is the unique identifier of the message and is assigned by the IPC. It is this *token* that is used by *SIGNAL* to indirectly deblock the sender. It is also used by the sender to later determine the status of a submitted message (e.g. still on a *path*, received, etc). Similarly, if a *RECEIVE* is issued without a *receive_point* specification in the *connect_point*, then the receiver is *blocked* until a message arrives for it.

SIGNAL is then simply of the form:

SIGNAL token

RECEIVE is similar in form to *SEND*:

RECEIVE token,to_port,path,from_port

where *(to_port,path,from_port)* refer to the message area to receive the message, the *path* (or indirection for the message), and the sending process name (optional). *Token* is the unique identifier of the transmitted message, returned upon successful completion of this operation.

If *path* is not specified in either *RECEIVE* or *CONNECT* then the *from_port* must be specified. In such a case, the receiver is asking for a message from a specific process and will either wait or continue asynchronously (in the event that a *receive_point* is specified in the *connect_point)*. On issuing a *RECEIVE*, a receiving process will get a message if a message is waiting in the IPC mechanism. If there is no message and there is no named *receive_point procedure* associated with the *CONNECT*, the process will be blocked. If there is a named *receive_point*, the process will be permitted to proceed asynchronously.

Similarly if the *from_port* is not specified on *RECEIVE* or *CONNECT*, then the *path* must be specified. *Path* identifies a queue or file that the receiver is willing to receive messages from any process using this *path*. The receiver will be able to receive messages sent to either this *path* or to the pair *path, to_port* = receiving process name. The receiver can not receive messages sent to the *path* and directed to another process, as a *path* can contain messages directed to more than one process from more than one process.

If a *from_port* and *path* are specified, then the receiver can receive messages sent to the *path* from only the specified process.

SUMMARY

What has been shown in the previous two sections is a simple set of primitives for process creation and inter-process communication.

The primitives are configuration independent and do not inhibit the installation from determining the appropriate logical systems structures.

There are many models of inter-process communications protocols which differ in the relation of *SEND/RECEIVE* to process blocking and concepts of WAIT, etc. They also differ in whether intervening mechanisms are visible to communicating processes, whether message collections survive process destruction, whether messages may be queued or forced upon receivers, and in conventions for the concept of reply and response.

This paper has described a design by which simple, direct, synchronous, transient interprocess communication may be undertaken without recoverability and integrity. As part of the same concept, an intervening file or queue may be imposed which allows many to many; one to many, one to any, interactions across protected paths. The concept of *SIGNAL* is a concept of response. Replies are seen to be undertaken through the issuance of sends at the convenience of a receiver when he wishes to respond in a meaningful way to a previous message.

The notion of *START* presented by this paper intends to provide a mechanism by which processes can initiate other processes and call for execution on nodes of the system that have various performance, status, load and scheduling attributes.

A paper under preparation discusses various aspects of the structure of an operating system that would support the language constructs discussed here.

REFERENCES

1. J.Baer "Multiprocessing Systems", *IEEE Transactions on Computers Vol.C25 No.12 (Dec. 1976)*
2. C.Hewitt, H.Baker *Actors and Continuous Functionals* MIT AI Memo 436 (Dec.1978)
3. J.B.Dennis *et al, Research Directions in Computer Architecture* MIT Report Number MIT/LCS/TM-114 (Sept.1978)
4. R.P.Goldberg "Survey of Virtual machine research" *Computer,* (June 1974) pp 34-35
5. J.J.Donovan, S.E.Madnick "Virtual machine advantages in security, integrity, and decision support systems" *IBM Systems Journal* Vol.15 No.3 (1976) pp 270-278
6. G.J. Meyers "Storage Concepts In a Software Reliability Directed Computer Architecture", *Proceedings of Fifth Annual Symposium On Computer Architecture,* IEEE, New York, 1979
7. P.Brinch Hansen *OPERATING SYSTEMS PRINCIPLES* Prentice Hall (1973).
8. J.K.Ousterhout, D.A.Scelza, P.S.Sindhu, "Medusa: An Experiment in Distributed Operating System Structure" *CACM 23,2* (Feb.1980) pp.95-105
9. E.I.Organick, A.I.Forsythe, R.P.Plummer, *PROGRAMMING LANGUAGE STRUCTURES* Academic Press (1978)

AN ARCHITECTURE FOR DIRECT EXECUTION OF REDUCTION LANGUAGES

Werner Kluge & Heinz Schlütter

Gesellschaft für Mathematik und Datenverarbeitung mbH Bonn
Postfach 1240, Schloß Birlinghoven
D-5205 St. Augustin 1

Abstract

A reduction language is a functional programming language whose semantics is defined by a set of rewrite rules.

Our paper describes the architecture of a machine which directly executes reduction language programs.

A laboratory model of this Reduction Machine has been built at the GMD Bonn and is currently used for experimental program design based on Berkling's version of a Reduction Language.

Introduction

Reduction language machines constitute a novel approach to computing that is radically different from the conventional von Neumann concept.

As the main feature of reduction languages is their strictly functional style of program design, the architecture of a Reduction Language Machine cannot be understood without having a basic knowledge of the language constructs and their execution.

There already exist a number of papers dealing with this subject, of which are primarily to mention those by J. Backus [BACKUS 72 & 78] and by K.J. Berkling [BERKLING 76] who originated the research in this field, and by F. Hommes [HOMMES 77 & 79] who implemented the first simulation model of a Reduction Machine.

However, it is thought helpful for the reader of this paper to be briefed on the Reduction Language with particular emphasis on the aspects that are relevant to an appropriate machine organisation.

The paper outlines a few basic Reduction Language constructs, their rules of execution, and the machine features that adequately support the processing of Reduction Language expressions.

Then we give an overview over the machine organisation and its operating principles, and a functional description of a hardware model of the Reduction Machine which has been constructed at the GMD [KLUGE 79].

The Reduction Language

As the Reduction Language is supposed to permit a strictly functional method of program design, its most fundamental construct is of the form

apply function to argument

The components of this expression map onto a binary tree with 'function' and 'argument' appearing in the left and right subtree, respectively, and with the 'apply to' as root node:

Fig.1

In general, 'function' and 'argument' are non-trivial tree-structured expressions. The 'apply to' is a constructor which relates two subexpressions in some meaningful way to each other.

More rigidly, an expression e of the Reduction Language is defined as e := con e1 e2, which is the preorder notation of the tree

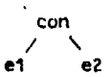


Fig.2

that links, by means of the constructor 'con', two subexpressions 'e1' and 'e2' to each other to form 'e'.

The most simple expressions are atoms, such as primitive function symbols, letter strings of any finite length representing variables, or strings of decimal digits which form decimal numbers.

Using this basic structure of Reduction Language expressions, a language designer would have to establish a set of primitive functions, data types and constructors, which must be complete in the sense that every computational problem can be formulated by a systematic application of these primitives.

In this paper, we do not discuss the development of such a complete language but introduce only a particular tree-processing primitive of a special

Reduction Language [HOHMES 79] to show the basic operating principle of the machine: let '>' be a constructor which builds binary trees, i.e. '> A B' is the tree

Fig.3

with 'A' as left and 'B' as right subexpression.

Let 'head' be a primitive function which selects the left subtree of such a binary tree, i.e.

apply head to > A B

results in 'A'; this transformation of an expression to another expression of the same meaning is called reduction.

Machine Organisation and Operating Principles

The basic machine functions that are necessary to execute Reduction Language expressions may be readily derived from what has been said about the language primitives in the previous section. Roughly speaking, there must be means to

- represent a Reduction Language expression in a suitable storage medium so that its tree structure is uniquely exhibited;
- perform a preorder traversal of the expression stored within this medium;
- recognize, within the immediate environment of the actual traversal position, the occurrence of a reducible subexpression;
- execute the reduction according to the meaning of the respective primitive expressions (which primarily involves traversal functions such as the comparison, deletion, insertion, and copying of subexpressions);
- resume, after the completion of a reduction, the traversal up to the topmost root node of the expression tree.

The first two problems were solved by representing the Reduction Language expressions in the preorder notation 'con e1 e2', and storing them in a push-down stack, with the root node symbol on the top: so, the expression-tree

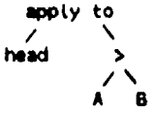


Fig.4

is represented as 'ap(ply) h(ea)d (to) > A B' in preorder and stored in a stack as

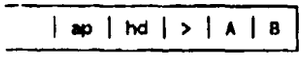


Fig.5

Since the preorder traversal scheme requires that the root node is inspected first, followed by the traversal of the left subtree in preorder, followed by the traversal of the right subtree in preorder, it simply takes a succession of pop-operations to have the expression emerge from the stack in the desired sequence, with the item on top of the stack being the actual traversal position.

A SINK-stack must be provided into which all symbols popped out of the first SOURCE-stack must be pushed in order to conserve the expression during the traversal. The expression ending up in the SINK-stack is supposed to appear with the root node symbols on top of its respective subexpressions. To accomplish this, a third stack is required as an intermediate storage for constructors since they emerge from the SOURCE-stack ahead of their subexpression but must enter the SINK-stack after them.

The corresponding traversal algorithm brings about the following phases with regard to the contents of the stacks E as SOURCE-stack, A as SINK-stack, and M as intermediate stack. Initially, the expression resides in the E-stack; the stacks A and M are empty, and the topmost item on E is inspected:

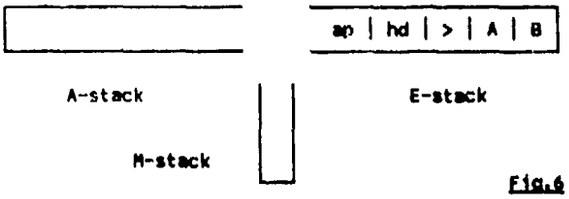


Fig.6

As the item is a constructor, it is transferred into the M-stack and marked with the superscript 'l' which indicates that the left subexpression of this constructor is now going to be moved from the E-stack to the A-stack:

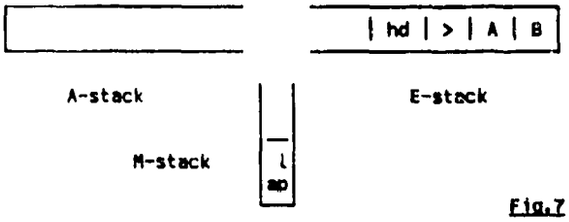


Fig.7

The focus of control returns to the top of the E-stack and moves the atom 'hd' into the A-stack:

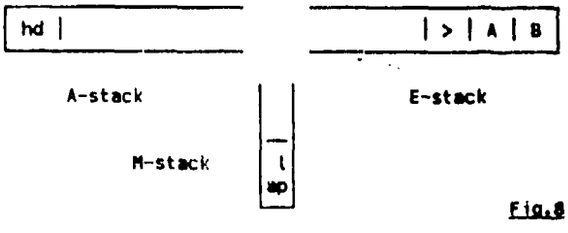


Fig.8

Then the focus of control turns to the M-stack. The 'ap' on top of the M-stack is found to be marked with an 'l'; as its left subexpression has just been moved over to the A-stack, the marking is changed to 'r', indicating that now its right subtree is on top of the E-stack:

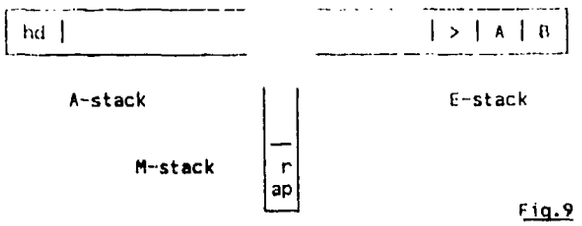


Fig.9

The top-element of the E-stack is a constructor '>' which is put into the M-stack and marked with an 'l':

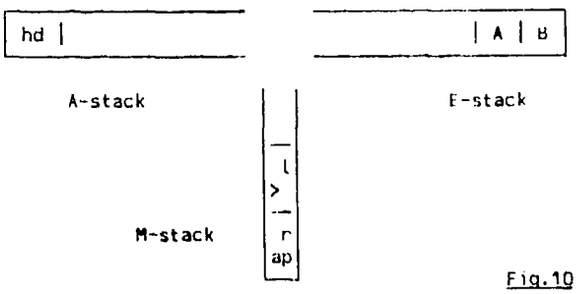


Fig.10

Then the left subtree 'A' of '>' is moved into the A-stack and the constructor '>' is marked with an 'r':

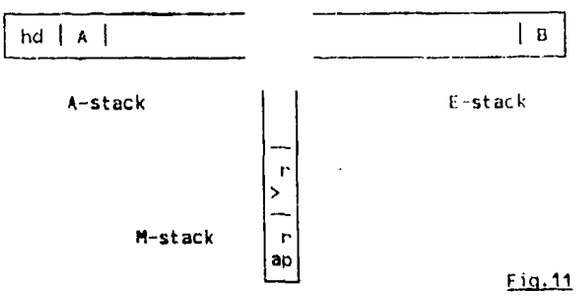


Fig.11

After the atom 'B' has been moved into the A-stack, the constructor '>' is found to be marked with an 'r' and can be pushed into the A-stack to complete the traversal of the subtree '> A B':

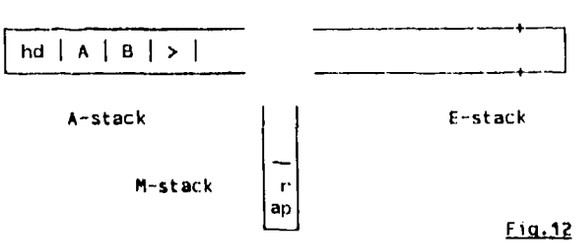


Fig.12

The constructor 'ap' which appears now on top of the M-stack is found to be marked with an 'r'. As its right subexpression has just been moved into the A-stack, the constructor 'ap' must be popped out of M and pushed into stack A:

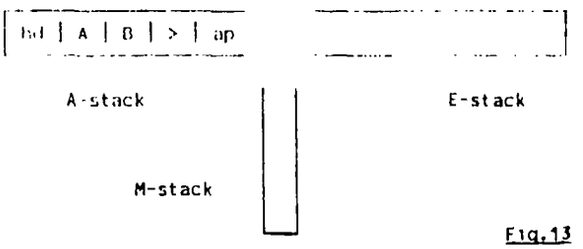


Fig.13

This completes the traversal since the stacks E and M are empty and the expression is lined up in the SINK-stack A in a transposed preorder form, with the left and right subexpression interchanged.

The execution of the same traversal algorithm with A as SOURCE- and E as SINK-stack reestablishes the original situation shown in Fig.6.

There are two important things that need to be noticed:

(1) The manipulation of the stack contents splits into two phases. First the item which constitutes the focus of control, the top of either the E-stack or the M-stack, is inspected. Then this item becomes the subject of a stack operation, which is either a transfer to another stack or a write-operation on the same stack.

(2) The constructor on top of the M-stack controls the movement of its subexpressions; moreover, there is a situation where 'ap' is on top of the M-stack, a function symbol is on top of the A-stack, and the argument expression is on top of stack E:

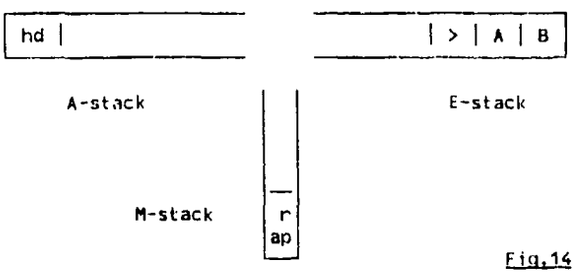


Fig.14

This property of the traversal scheme serves to recognize reducible expressions.

In our example, the traversal scheme brings about a situation where 'ap' appears on top of stack M and the function 'hd' on top of stack A.

This situation may be readily detected by simultaneously watching the tops of the E-, A- and M-stack during the execution of the preorder traversal.

If an instance of a reduction rule occurs, the traversal is immediately suspended and control switches to another algorithm which performs the appropriate reduction steps.

The reduction algorithm calls other algorithms which participate in the evaluation of the particular subexpression.
This transfer of control is accomplished by conventional methods of subroutine stacking: code words representing the algorithms are, in their order of activation, pushed into a system control stack S, and popped out upon termination so that control eventually returns to the original traversal algorithm.

The reduction of an expression involves rather simple primitive operations like the deletion of an expression which may be viewed as a traversal without a SINK-stack, copying which is a traversal with one SOURCE-stack and two SINK-stacks, and comparison which is a traversal with two SOURCE-stacks and one SINK-stack.

For instance, the reduction of the expression in Fig.14 can be done as follows: first, the primitive function 'hd' in the A-stack, the constructor 'ap' in the M-stack and the tree-constructor '>' on top of the E-stack are deleted; then the atom 'A' is moved to the A-stack, the atom 'B' is deleted and 'A' is moved back into the E-stack; so, 'apply head to > A B' is reduced to 'A'.
Of course, this procedure also works properly if 'A' and 'B' are not only atoms but trees.

Other important algorithms include those for performing arithmetic operations on decimal numbers of any finite length. In this case, two atomic subexpressions representing the operands must, symbol by symbol, be popped out of their respective SOURCE-stacks and moved through an arithmetic unit whose output is pushed into the SINK-stack.

To provide sufficient space for expression manipulation it is convenient to have more than the stacks E, A, M and S available: so, the machine has another three stacks named B, U and V to store expressions.

An expression is manipulated only by push, pop, read or write operations affecting the items residing on top of the stacks.

There is no addressing of objects within the expression involved: they may become the focus of attention only through an orderly traversal of the expression tree which brings them to the top of one or the stacks. Addresses are used only to identify the stacks that are to be operated upon in a particular instance.

Control over the stacks is exercised by means of the Reduction Unit which may be considered as the processing unit of the machine. The overall function of the Reduction Unit is very simple. Under the control of the algorithm residing on top of the system control stack S, it inspects the topmost symbols of one or two selected stacks. Thereupon, it goes through a decision process (realized by combinatorial logic networks) as a result of which it may issue new symbols and specify stacks which are to be pushed, popped, written into and read next. A small sequential network, comprising some status flipflops, navigates the machine through the sequence of actions required by the reduction process.
More specifically, the Reduction Unit provides all the facilities to perform the various traversal algorithms, to recognize instances of reductions, and to execute the reductions, including an arithmetic unit for arithmetic operations on decimal numbers.
There is also an I/O-Processor which loads expressions into the machine and unloads them after reduction, and via which the user may exercise control over the machine.

An elementary cycle of operation within the Reduction Machine quite naturally partitions into four phases as illustrated below:

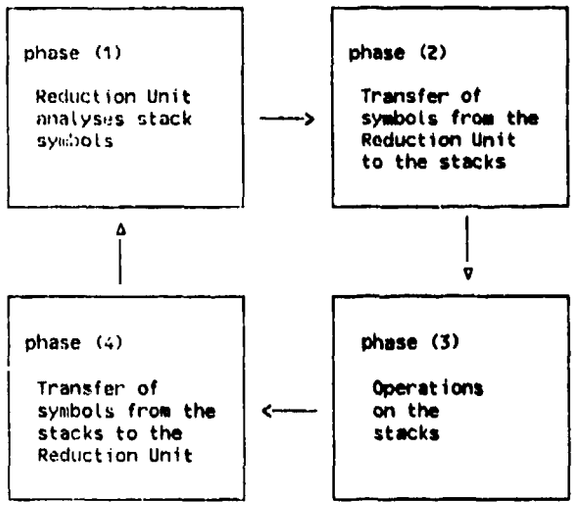


Fig.15

Starting in phase (1), the Reduction Unit is about to analyse what it has just read from the selected stacks. Then the machine enters phase (2) during which push, pop, read and write control signals, together with new symbols, are transferred from the Reduction Unit to the stacks.
During phase (3), up to four stacks can be pushed and popped such that new symbols appear in their topmost positions at the end of this phase. During phase (4), the topmost items of the stacks which have been selected for a read operation are moved into the Reduction Unit which again enters phase (1).

The GMD Hardware Model

The hardware model of the Reduction Machine was primarily intended to demonstrate the feasibility of the Reduction Language principles. Its design was largely determinded by the objective of getting a simple and reliable machine into operation as quickly as possible.

The machine employs standard low-power Schottky TTL technology for all logic circuits, registers, status-flipflops etc., fast read-only-memory devices for the realization of a control store in which the reduction algorithms are implemented, and dynamic random access memory chips for the realization of the stacks.

All machine operations are under the control of a central clock which subdivides a machine cycle into eight intervals of equal length. As the clock runs at a frequency of 6.25 MHz, an interval lasts 160 nsec and a machine cycle lasts 1.280 microseconds. The effective speed of operation, however, is slightly slower since every 16th machine cycle is used for a refresh operation on all stacks.

A block diagram of the hardware architecture is shown in Fig.16. It comprises the Reduction Unit (which is subdivided into four subunits named TRANS, REDREC, REDEX, ARITH), a set of seven pushdown stacks, a bus system which handles the traffic of symbols and control-signals between the Reduction Unit and the stacks, a central timing system CTS, and an I/O-Processor (a conventional INTEL SBC 80/20 single board computer) which also performs some monitoring and preprocessing functions.

The data paths within the entire machine are laid out to accommodate byte formats (eight bits plus parity), i.e. all stacks, data busses and Reduction Unit circuits are one byte wide.

The Reduction Unit comprises four modules, each of which is accommodated by a separate printed circuit board:

- TRANSport performs all traversal algorithms (including deletion, comparison, copying);
- REDuction RECognition is a combinatorial logic network that looks, during the traversal of an expression, for the appearance of an instance of a reduction. Upon the detection of a reducible expression, REDREC immediately deactivates the TRANS-subunit, pushes a new algorithm-code on top of the S-stack and turns control over to
- REDuction EXecution, which essentially comprises a fast control memory containing all the control programs which are required to perform the reductions. As for arithmetic operations, REDEX is supported by the
- ARITHmetic unit which performs the arithmetic operations on the decimal numbers which, under the control of REDEX, are received digit by digit from the respective SOURCE stacks; the resulting digits are sent back to the SINK stack.

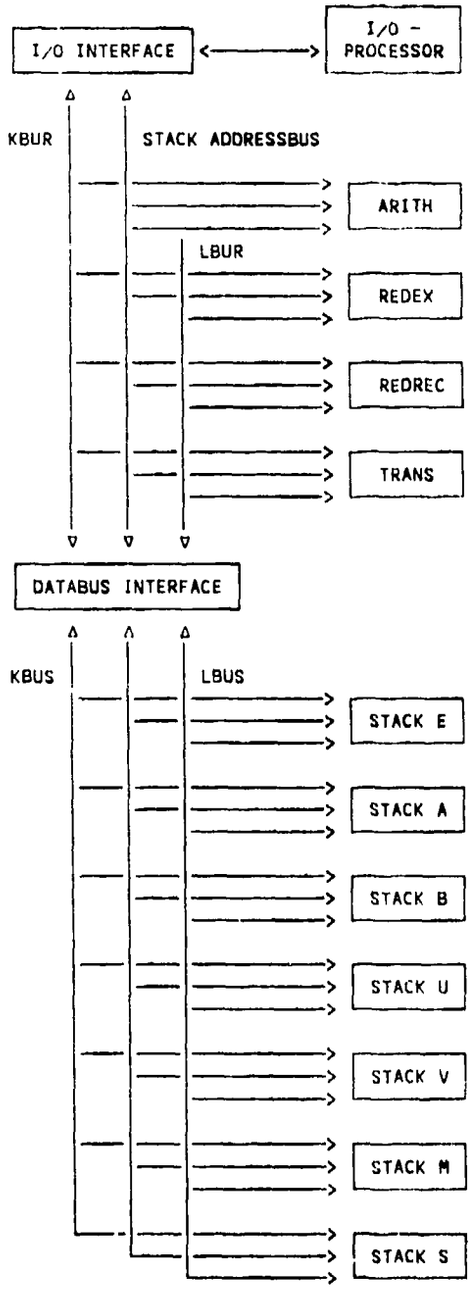


Fig.16

Block Diagram of the Reduction Machine Architecture

A stack is schematically shown in Fig.17. The major components are the random access memory, a stack pointer to the actual top-of-stack location, a separate TOP OF STACK register in which the actual topmost item resides, and a COPY-register in which a copy of the contents of the TOP-register is held.

The TOP-register may receive a data item, via the multiplex circuit INBUSSELECT, from one of three sources: the KBUS, the LBUS, or the memory cell that is addressed by the stack pointer. The contents of the TOP-register may be supplied to the KBUS or LBUS via the multiplex circuit OUTBUSSELECT.

The stack operations are as follows: the TOP-register contains the topmost data item k of the stack, a copy of which is in the COPY-register; the stackpointer addresses the first empty cell of the random access memory stack area.

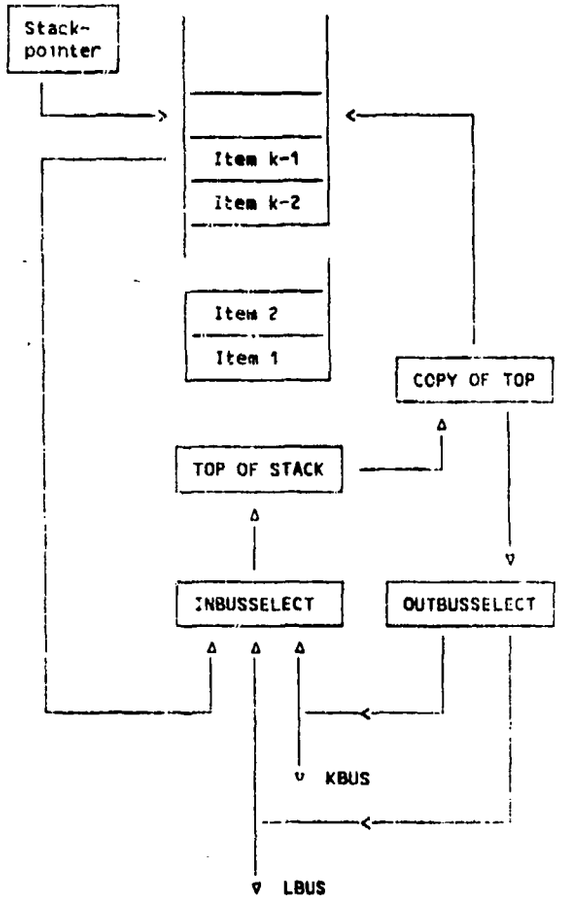


Fig.17

Block Diagram of the Reduction Machine Stack Organisation

Upon a push operation, an item enters via INBUSSELECT from KBUS or LBUS and is written into the TOP-register. Subsequently, the contents of the COPY-register, i.e. the old top of the stack, are stored away in the empty cell addressed by the stackpointer. Afterwards, the stackpointer is incremented by one to point again to the first empty cell, and the contents of the TOP-register are copied into the COPY-register.

Conversely, if the stack is to be popped up, the stackpointer is first decremented by one to point to the last occupied cell, then the contents of this memory cell are read out and written into the TOP-register, whose new value is copied into the COPY-register.

Read and write operations affect only the contents of the TOP- and COPY-registers and cause no memory access cycle.

Input/output processing and certain system support functions are handled by a conventional INTEL SBC 80/20 single board microcomputer which, via a tailor-made I/O-interface, is attached to the bus system of the Reduction Machine. The currently implemented I/O-configuration only supports a data station Hewlett Packard HP 2645A which perfectly suits the purpose of the Laboratory Model: Reduction Language expressions can be edited, shipped into the Reduction Machine for the execution of a user-specified number of reductions, and displayed afterwards. As the HP 2645A data station includes two tape cartridge drives, user expressions and standard library functions may be stored away to and retrieved from tape.

Perspective

When assessing its strengths and weaknesses, the Reduction Machine architecture and its hardware realization as described in this report should be seen in the light of the following aspects:

- the Reduction Machine is the first of its kind that directly supports the execution of reduction languages; its architecture has been straightforwardly derived from the basic structure of reduction language expressions and their rules of execution;
- the concept of not using addresses for the representation of expressions within the Reduction Machine has nowhere been compromised;
- the Reduction Machine was primarily conceived as an interactive tool for systematic construction of functional programs, serving only one user at a time;
- the hardware model was simply intended as a vehicle to demonstrate that the Reduction Language concept can be adequately supported by the proposed machine architecture; neither memory capacity nor performance in terms of program runtime were a design objective. s()

There remain a number of problems that need to be solved before the Reduction Machine can be accepted as a competitive alternative to von Neumann computers. At the level of machine architecture, these problems concern

- adequate interfacing with peripheral memory devices like disks and tapes to support program libraries, serious data base applications, and also the concept of 'virtual stacks', i.e. transparent stack extension into secondary storage;
- program-controlled input and output of expressions from and to peripheral devices;
- interrupt facilities supporting the communication with I/O-Processors, real time applications and the cooperation with other Reduction Machines;
- measures that remedy a serious performance degradation in list processing applications which is caused by excessive copying activities.

Preliminary studies have shown that program controlled I/O and interrupt handling can neatly be integrated into the language concept by introducing appropriate constructs.

As it appears now, the interfacing with conventional peripherals necessitates traditional file management methods and data transmission techniques since device controllers are designed for standard interfaces with conventional computers. Hence, the microprocessor approach for I/O-handling which has been taken with the Laboratory Model seems to be a step into the right direction, guided by the type of peripheral devices that are currently available in the market-place. However, with future advances in electonic disk technologies, stack-type peripheral memory devices of sufficiently large capacity that are compatible with the internal structure of the Reduction Machine may be anticipated.

To significantly expedite the processing of large list structures, the hard-lined 'no-addresses' approach may have to be softened to some extent. Conceivably, subexpressions could be linked to their respective constructors by relative pointers within the internal representation of an expression. Along these pointers, the focus of control could be moved directly to a particular subexpression rather than traversing linearly through the expression tree that is to the left and above it.

It may also be envisaged that such a pointer structure facilitates the partitioning of an expression into subexpressions of suitable meaning that can be distributed for concurrent processing within a system of cooperating Reduction Machines.

References

[BACKUS 72] Backus, J.
Reduction Languages and Variable-free Programming
IBM Research Report RJ 1010, April 7, 1972

[BACKUS 78] Backus, J.
Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs
CACM 21, No.8, Aug. 1978, pp.613-641

[BERKLING 76] Berkling, K.J.
Reduction Languages for Reduction Machines
Interner Bericht ISF-76-8, 1976
GMD, Schloß Birlinghoven, D-5205 St.Augustin 1

[HOMMES 77] Hommes, F.
The Internal Structure of the Reduction Machine
Interner Bericht ISF-77-3, 1977
GMD, Schloß Birlinghoven, D-5205 St.Augustin 1

[HOMMES 79] Hommes, F. & Schlütter, H.
Reduction Machine System User's Guide
GMD, Schloß Birlinghoven, D-5205 St.Augustin 1

[KLUGE 79] Kluge,W.
The Architecture of a Reduction Language Machine
Hardware Model
Interner Bericht ISF-79-3, 1979
GMD, Schloß Birlinghoven, D-5205 St.Augustin 1

AN EXPRESSION ORIENTED EDITOR FOR LANGUAGES WITH A CONSTRUCTOR SYNTAX

Ferdinand Hommes

Gesellschaft für Mathematik und Datenverarbeitung mbH Bonn
Postfach 1240, Schloß Birlinghoven

Abstract

A wide class of languages can be defined by using a constructor syntax. This paper gives a short introduction to the constructor syntax and describes an interactive editing system for languages having such a syntax. In contrast to conventional line-oriented editors this editing system is completely expression-oriented. The system has been successfully implemented for Berkling's Reduction Machine.

I. Languages with a Constructor Syntax

1. Definition of a Constructor Syntax

Backus introduces in his report [BACKUS 73] languages with a constructor syntax: The pair (A,K) is a constructor syntax for a language E if the following conditions hold:

1. $A \subset E$
2. Each $k \in K$ is a function from a subset Sk of E^n into E
3. For every $e \in E$, either $e \in A$ or there are a unique $k \in K$ and unique $e_1,\ldots,e_n \in L$ such that $k[e_1,\ldots,e_n] = e$.

Each element a of A is called an atom, and each $k \in K$ is called a constructor. Let $k[e_1,\ldots,e_n] = e$, then $e_1,\ldots,e_n$ are called subexpressions of the expression e and k is called an n-place-constructor. Each expression of a language with a constructor syntax is either an atom or can be written as $e = k[e_1,\ldots,e_n]$.

Example 1: Definition of a language

Let $A = (a_1,..,a_5)$ and $K = (k_1,k_2)$ with $ki \in [ExE \longrightarrow E]$, i.e. the ki's are two-place-constructors. Then (A,K) is a constructor syntax defining a language which we call E and to which we will refer in the following chapters. An example of an expression of the language E is $k_1[a_1,k_1[a_2,k_2[a_3,k_2[a_4,a_5]]]]$

Expressions of languages with a constructor syntax can be represented as trees. Atoms become the leaves of the trees, whereas the constructors form the nodes.

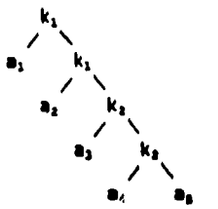


Fig. 1: Tree-representation of the expression of example 1.

The language E which has been defined in Example 1 looks very abstract, for we did not associate any meaning with the atoms or constructors, we just gave them formal names.

Now we are going to discuss the following two representations of the language:

1. Its representation within a machine (machine interface or internal representation)
2. Its representation on a display station (user interface or external representation)

2. Internal Representation

A representation of an expression within a given machine is obtained by:

1. coding the atoms and constructors
2. mapping the structure of the expressions into storage

Each atom or constructor is stored within a memory cell; the coding function maps the symbolic name of an atom or constructor into a value which fits into a memory item, e.g. a_5 is mapped to the hexadecimal constant X'35'.

In the following we will denote the coding of an atom or a constructor x by $x, i.e. the symbolic name for the coding of the atom a_5 is $a_5.

We have already mentioned that each expression can be represented by a tree; this means that we have to map a tree-structure into memory. A convenient way to do that is to connect the elements by pointers. Figure 2 shows such a realization for the expression defined by Example 1:

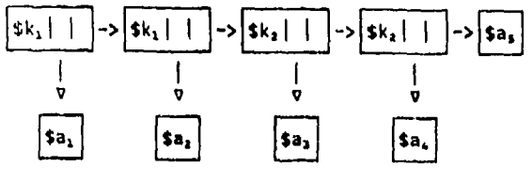


Fig. 2: Representation of an expression by using pointers.

Berkling has used another method within his Reduction Machine: the expressions are stored within stacks, using the preorder notation of the associated expression-tree. Figure 3 shows how the expression of Example 1 is stored.

Top of stack

k_1	a_1	k_1	a_2	k_2	a_3	k_2	a_4	a_5

Fig. 3: Representation of an expression in a stack using preorder notation.

In this paper we will prefer the stack representation since it has the following advantages:

1. The representation is very close to the formal definition of expressions, i.e. removing brackets and commas from the formal definition leads directly to the preorder notation (cf. Example 1 and Figure 3).

2. It is free of pointers which are not directly related to the problem.

Thus the algorithms of the editing system which we are going to describe will be more clear and precise, for they are free from pointer manipulation and garbage collection problems.

3. External Representation

Normally the user is not interested in the internal coding of an expression. He wants to see certain keywords or strings which have a meaning to him.

Therefore we need another function - the I/O-function - mapping formal expressions into expressions which can be understood by the user. The I/O-function can be defined by a table which associates all atoms with a string and all constructors with a prototype-expression that consists of some keywords and place-holders (□) which indicate where the subexpressions are going to be inserted. In Figure 4 a possible I/O-table for the language defined in Example 1 is shown:

a_1 = head	a_2 = tail	a_3 = A	a_4 = B	a_5 = C
k_1 = apply □ to □		k_2 = > □ □		

Fig. 4: I/O-table for the language given by Example 1 (translation to Berkling's Reduction Language).

Using the I/O-table above the expression of Example 1 is displayed as:

```
apply head
   to apply tail
        to > A > B C
```

which is a valid expression for Berkling's Reduction Language.

Different I/O-tables may exist for the same formal language. The next figure shows a translation of formal expressions to LISP:

a_1 = CAR	a_2 = CDR	a_3 = A	a_4 = B	a_5 = C
k_1 = (□ □)		k_2 = (□ . □)		

Fig. 5: I/O-table for the language given by Example 1 (translation to LISP).

Using this table results in: (CAR(CDR(A.(B.C))))

The editing system which we are going to develop will only be based on the formal definition of expressions. The external representation of an expression is generated by using an I/O-table, which may be a default table supplied by the system, or a table defined by a user who wants to use his own external representation of a language.

The next figure shows the relationship between the different representations of an expression:

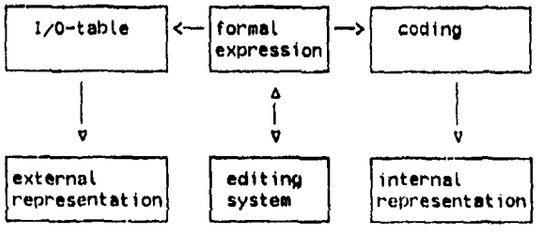


Fig. 6: Representations of formal expressions.

II. The Interactive Editing System

1. An Expression Oriented Editing System

Conventional editors are line-oriented, i.e. a line is the smallest logical unit. Almost all commands of such an editor refer to lines, e.g. move lines, copy lines, insert lines, scroll up and down a cer-

tain number of lines. There is no relationship to the structure of the program which is edited, or to the language it is written in. If a user wants to delete a begin-end-block, he has to find the corresponding lines for deletion. This can be very tedious if nested blocks are used or the block does not fit onto the display.

We have implemented an editor which is not line-oriented but expression-oriented, i.e. the smallest logical unit the user can handle is a complete expression. All commands of the editor will refer to expressions, e.g. copy expressions, move expressions, delete expressions, scroll to a subexpression etc.

2. Algorithms to Handle Expressions

Since the smallest logical units in our editing system are complete expressions, we first describe some basic algorithms which allow us to move, copy and to delete expressions.

The editor works with five stacks which are called E, A, M, B, and U. Expressions are stored within stacks using the representation described in I.2 (cf. Figure 3).

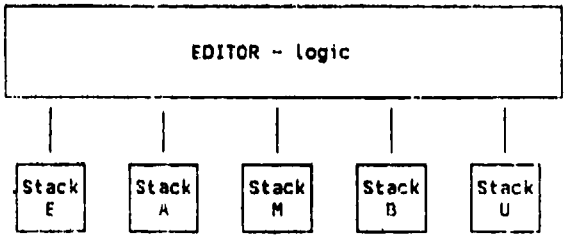


Fig. 7: Memory used by the Editor.

There are the following primitive procedures and functions to handle stackelements:

POP(X): deletes the item on top of stack X

PUSH(I,X): pushes item I into stack X

MOVE(X,Y): moves one item from stack X to stack Y

MOVE2(X,Y,Z): moves one item from stack X to the stacks Y and Z.

The functions and procedures listed above will not be explained any further in this paper.

2.1. The Algorithm TRANSPORT

The algorithm TRANSPORT moves a complete expression from one stack to another stack. A third stack, the control-stack M, is used for intermediate saving of constructors. A call of the algorithm TRANSPORT is denoted by TRANSPORT(X,Y) where X and Y are stacknames and the expression is moved from the stack X to the stack Y, i.e. TRANSPORT(E,A) moves an expression from stack E to stack A.

In the following we will use a PASCAL-like language to specify algorithms. We assume that there is a global type-declaration of the stacks:

```
TYPE STACKNAME = (E,A,M,B,U);
```

Then the algorithm TRANSPORT is given by

```
PROCEDURE TRANSPORT(X,Y:STACKNAME);
BEGIN
  CASE TOP(X) OF
            ATOM: MOVE(X,Y);
   N-CONSTRUCTOR: BEGIN
                    MOVE(X,M);
                    FOR I:=1 TO N
                     DO TRANSPORT(X,Y);
                    MOVE(M,Y);
                  END
  END
END
```

TRANSPORT is a recursive algorithm: after a constructor has been saved in the M-stack all its subexpressions are moved to the sink-stack, then the constructor is moved from the M-stack to the sink-stack.

Note: During the transport the subexpressions of constructors are interchanged, e.g. applying the algorithm TRANSPORT to the expression shown in Figure 3 yields:

Top of stack

Fig. 8: Result of transporting the expression given by Figure 3.

Applying the algorithm TRANSPORT repeatedly to an expression yields the following transformation:

$$K(e_1,\ldots,e_n) \xrightarrow{\text{TRA.}} K(e_n,\ldots,e_1) \xrightarrow{\text{TRA.}} K(e_1,\ldots,e_n)$$

i.e. an even number of transports always yields the original expression.

2.2. The Algorithm TRANSPORT2

The algorithm TRANSPORT2 moves an expression from one stack to two other stacks. It is called by TRANSPORT2(X,Y,Z), where X, Y, and Z are stacknames; Z denotes the second stack to which the expression is moved. The algorithm differs in only one point from the algorithm TRANSPORT: atoms and constructors are pushed into two sink-stacks.

```
PROCEDURE TRANSPORT2(X,Y,Z:STACKNAME);
BEGIN
  CASE TOP(X) OF
            ATOM: MOVE2(X,Y,Z);
   N-CONSTRUCTOR: BEGIN
                    MOVE(X,M);
                    FOR I:=1 TO N
                     DO TRANSPORT2(X,Y,Z);
                    MOVE2(M,Y,Z);
                  END
  END
END
```

2.3. The Algorithm COPY

This algorithm copies an expression from one stack to another stack without interchanging the sub-expressions. COPY has the same parameters as the algorithm TRANSPORT, i.e. COPY(A,E) copies an expression from stack A to stack E. nze() The algorithm COPY uses the algorithms TRANSPORT2 and TRANSPORT:

```
PROCEDURE COPY(X,Y:STACKNAME);
VAR Z1,Z2: STACKNAME;
BEGIN
  Z1:= ...; Z2:= ...;
  TRANSPORT2(X,Z1,Z2);
  TRANSPORT(Z1,X);
  TRANSPORT(Z2,Y);
END
```

The stacks Z1 and Z2 are used as scratch pad stacks for expressions. They must be different from the stacks X and Y.

2.4. The Algorithm DELETE

The algorithm DELETE removes a complete expression from a stack. It has only one parameter which is the name of the stack where the expression is to be deleted, i.e. DELETE(E) deletes an expression in stack E:

```
PROCEDURE DELETE(X:STACKNAME);
BEGIN
  CASE TOP(X) OF
           ATOM: POP(X);
   N-CONSTRUCTOR: BEGIN
                    POP(X);
                    FOR I:=1 TO N DO DELETE(X);
                  END
  END
END
```

The algorithms described above have been implemented by hardware in Berkling's Reduction Machine. A description is given in [KLUGE 79].

3. The Use of the Different Stacks

We have already mentioned that the editor works with five stacks which are called E, A, M, B, and U. Stack E contains the expression which is displayed to the user. We call this expression Focus of Attention (FA). Stack M is the control stack which is used by the TRANSPORT-algorithm. Stack B is used for input and output, i.e. input operations move an expression from the display station to stack B whereas output is done by moving an expression from stack B to the display. Stacks A and U are used by the scrolling operations.

4. Output of Expressions

Output of an expression means: Display the current Focus of Attention. First of all the expression in stack E is copied to stack B, from where it is moved by the algorithm OUTPUT to the display buffer of the terminal.

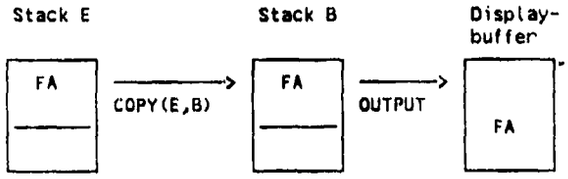


Fig. 9: Output of an expression.

The algorithm OUTPUT is a modified TRANSPORT-algorithm. It is defined by

```
PROCEDURE OUTPUT;
BEGIN
  CASE TOP(B) OF
            ATOM: DISPLAY;
   N-CONSTRUCTOR: BEGIN
                    DISPLAY;
                    FOR I:=1 TO N DO OUTPUT;
                  END
  END;
  ERROR: BEGIN
           DELETE(B); ABBREVIATE;
         END;
END
```

The procedure DISPLAY pops one item out of stack B, retrieves its representation from the I/O-table (cf. I.3), and replaces the associated placeholder within the display-buffer by the representation. Before the algorithm OUTPUT is called, the display buffer is cleared and one placeholder is inserted. Figure 10 shows the different states of the algorithm OUTPUT, using the I/O-table given in Figure 4 and the expression $k_1[a_1,k_2[a_4,a_5]]$:

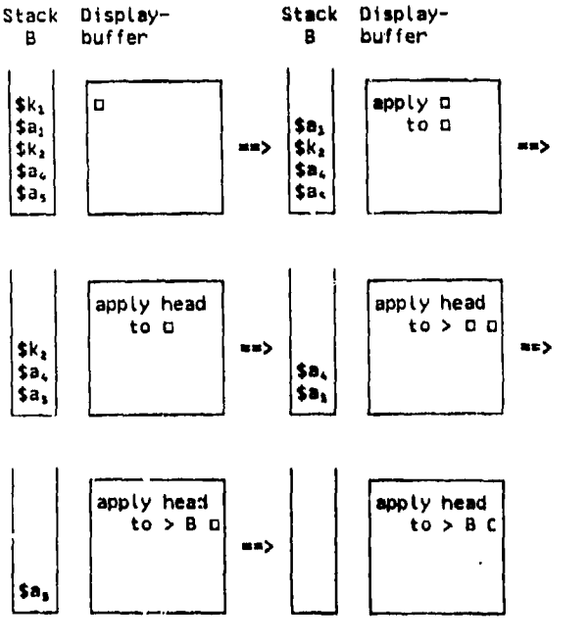


Fig. 10: Example showing the different phases of algorithm the OUTPUT.

The procedure DISPLAY fails if there is not enough space within the display buffer to insert the representation of an atom or a constructor. In this case the error exit is taken: The corresponding subexpression in stack B is deleted and the algorithm ABBREVIATE replaces the current placeholder by an abbreviation symbol. This means that the innermost subexpressions are automatically abbreviated and the complete Focus of Attention is shown on the display.

5. Scrolling and Displaying Selected Subexpressions

In this chapter we will describe how the user can change the Focus of Attention in order to look at subexpressions which have been abbreviated. Line-editors can display hidden information by means of scrolling commands: Display previous page, display next page, scroll up n lines etc., i.e. scrolling is completely line-oriented. For our purpose we need a scrolling mechanism which is expression-oriented since the hidden information always consists of complete subexpressions.

But the problem is how to select a subexpression on the display and how to find the corresponding subexpression within the expression in the E-stack. The solution we are looking for should be independent of the current I/O-table that is used for displaying expressions; it should only depend on the constructor syntax.

An easy way to select a subexpression is to move the cursor to its position on the display. But cursor-addresses are not expression dependent, they are just given by a line and column number. So we have to translate the cursor-address into an appropriate expression-address. In our editing system these 'appropriate' addresses are themselves expressions taken from a special address language LADDR. The language LADDR is defined by the following constructor-syntax:

Let A = (1,2,...} U (nil), i.e. an atom is either a natural number or nil, and let K = (K2ADDR) where K2ADDR is a two-place constructor. Then the editor will use the following expression of LADDR as address of expressions or subexpressions:

1. The root of an expression gets the address nil
2. The i'th subexpression gets the address K2ADDR[I,ADDR] where ADDR is the address of the current expression

In order to make addresses more readable we use the following representation for the constructor K2ADDR: K2ADDR[X,Y] = X.Y

The next figure shows the expression of Figure 10, where each subexpression has been marked with its address.

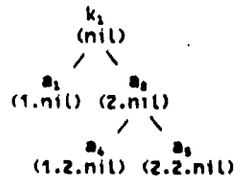


Fig. 11: Expression and its addresses.

Note: All addresses terminate with the special atom nil. Readers who are familiar with LISP probably have noticed that expression-addresses are represented by a list of integers. As expression-addresses are based on a constructor syntax, the basic algorithms TRANSPORT, COPY, etc. may also be applied to them. Besides these we will need some other algorithms to handle addresses:

HEAD(ADDR): extracts the first number from an address, i.e. HEAD(1.2.3.4.nil)=1

TAIL(ADDR): removes the first number from an address, i.e. TAIL(1.2.3.4.nil) = 2.3.4.nil

REVERSE(ADDR): reverses the sequence of the numbers that constitute an address,i.e. REVERSE(1.2.3.4.nil) = 4.3.2.1.nil

These algorithms can be expressed by using the basic transport algorithm and the operations POP, PUSH, and MOVE.

Given a reversed expression-address, we can define an algorithm SCROLLDOWN which selects the corresponding subexpression. Basically, SROLLDOWN is a transport-algorithm which moves an expression from stack E to stack A, but the transport is stopped as soon as the selected subexpression is on top of stack E:

```
PROCEDURE SCROLLDOWN(ADDR: EXPRESSION-ADDRESS);
BEGIN
  IF   NOT(ADDR = nil)
  THEN BEGIN
         MOVE(E,M);
         WHILE I < HEAD(ADDR)
          DO BEGIN TRANSPORT(E,A); I:=I+1; END;
         SCROLLDOWN(TAIL(ADDR));
       END
END
```

When the SCROLLDOWN algorithm stops, the stacks A and M contain the environment of the selected subexpression. Stack M contains all the constructors which have been encountered when walking to the subexpression, whereas stack A contains all the subexpressions which have been removed in order to get the subexpression on top of stack E.

After having selected a subexpression and after having performed some actions on it the user may want to return to the expression from where scrolling was invoked. This is done via the algorithm SCROLLUP which is the inverse of the algorithm SCROLLDOWN:

```
PROCEDURE SCROLLUP(ADDR: EXPRESSION-ADDRESS);
BEGIN
  WHILE I < HEAD(ADDR)
   DO BEGIN TRANSPORT(A,E); I:=I+1; END;
  MOVE(M,E);
  SCROLLUP(TAIL(ADDR));
END
```

Before the algorithm SCROLLUP is called the expression-address is not reversed. SCROLLUP moves the subexpressions and constructors having been moved to the stacks A and M back to stack E, thus

reconstructing the original expression again.

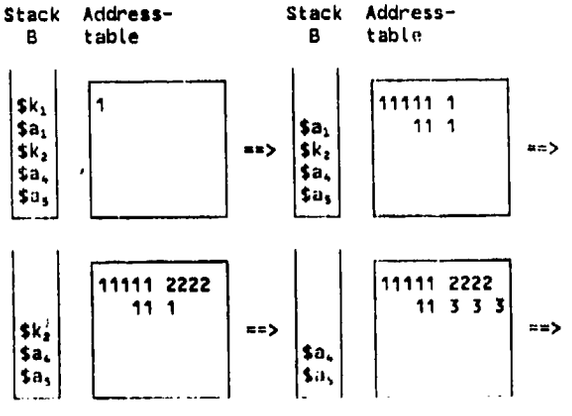
The editing system also supports nested scrolling: Whenever the algorithm SCROLLDOWN is called the associated expression-address is moved to stack U. A sequence of scroll-downs then generates a sequence of expression numbers within stack U. When scroll-up is requested the required expression-address is found on top of stack U from where it is removed.

Now there is one problem left: Which expression-address belongs to which cursor-address? This relationship is established via algorithm OUTPUT which is extended in the following way: For each atom and for each constructor the corresponding expression-address generated:

```
PROCEDURE OUTPUT(ADDR: EXPRESSION-ADDRESS);
BEGIN
  CASE TOP(B) OF
            ATOM: DISPLAY
    N-CONSTRUCTOR: BEGIN
                     DISPLAY;
                     FOR I:=1 TO N
                      DO OUTPUT(I.ADDR);
                   END
  END
  ERROR: BEGIN
           DELETE(B);
           ABBREVIATE;
         END
END
```

When algorithm OUTPUT is called for the first time ADDR should be nil, i.e. OUTPUT(nil) is a valid call.

The procedure DISPLAY of algorithm OUTPUT has to be extended, too. First of all we need in addition to the display buffer a second buffer which we will call address table. The address table has as many entries as characters can be displayed on the display. Each entry contains the address of an expression-address. Now, the procedure DISPLAY will update both, the display buffer and the address table: Whenever the representation of an item is moved to the display buffer, the corresponding entries within the address table will receive the address of the current expression-address. Figure 12 shows the contents of the address table for the different phases of algorithm OUTPUT for the example given in Figure 10.



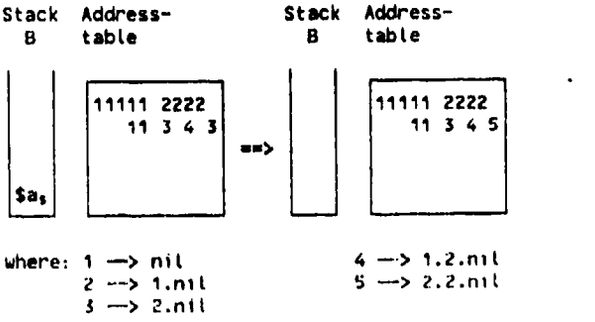


Fig. 12: Contents of the address table for Figure 10.

No entry means that at the corresponding position of the display no expression is shown. The algorithm ABBREVIATE will insert the address of the expression number of the abbreviated expression into the address table.

Now we are able to translate a cursor-address into an expression-address: The cursor-address denotes an offset within the address table, where we find the address of the associated expression-address:

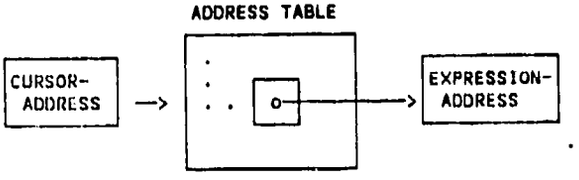


Fig. 13: Association of cursor- and expression-address

The existence of an address table allows an expression oriented use of some standard display-station keys, e.g. the EOF-key (= Erase until end Of Field) can be changed to a more useful EOE-key (= Erase until end Of Expression). An expression is erased by erasing all screen-positions whose expression-addresses have the same suffix as the current address of the expression.

6. Editing: Update of Expressions

Until now we have described the passive part of the editing system, e.g. the representation of expressions, how they are displayed etc. Now we turn our attention to the active part of the system which allows the user to edit (= update, delete, replace, etc.) expressions.

6.1. Format of the Screen Image

First of all we have to specify the screen image used by the editor. The screen of a display should be divided into four logical parts as shown in Figure 14.

EXPRESSION-field	FA-field
MESSAGE-field	COMMAND-field

Fig. 14: Logical fields used by the editing system

The FA-field is the area in which the current Focus of Attention is displayed via algorithm OUTPUT. In the COMMAND-field the user may specify editor-commands. The MESSAGE-field is used to display additional information like error messages, explanations of the commands etc.

The EXPRESSION-field is used to update expressions. Figure 15 shows how the four fields can be mapped onto the screen of a real display-station. This display image is used by the editing system of Berkling's Reduction Machine.

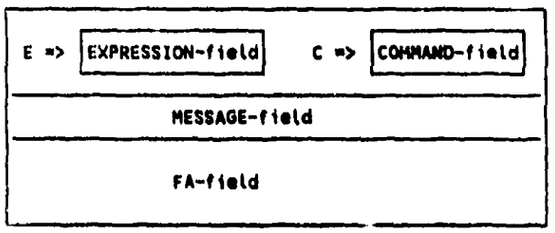


Fig. 15: Display image used by the editing system of Berkling's Reduction Machine.

6.2. Editor-Commands

We have already mentioned that the Focus of Attention, i.e. the expression which resides on top of stack E, is displayed within the FA-field of the screen. Now let us consider a subexpression of FA which is given by the current cursor-position. We will call this expression the CURSOR-expression (CE) and denote its address by CEADDR.

All editor commands refer to CE, this means that CE must be on top of stack E when the specified command is going to be executed. Thus we have to perform a SCROLLDOWN before and a SCROLLUP after the execution of a command:

```
SCROLLDOWN(REVERSE(CEADDR))
execute specified monitor command
SCROLLUP(CEADDR)
```

In this paper we will give only the description of two basic editing commands:

D: Delete the CURSOR-expression
Cx: Copy the CURSOR-expression

The D-command replaces CE by a special atom called the EMPTY-expression which prompts the user to enter a new expression. This ensures that a user can never generate incomplete expressions. Whenever he deletes an expression he has to replace it by another expression. The algorithm for the D-command is:

```
DELETE(E); PUSH(EMPTY-EXPRESSION,E);
```

The copy-command copies CE either to an auxiliary stack (x = STACK0, STACK1, etc.) or into an expression library (x = name of an expression). Copying is done in the following way: At first the expression is copied to the I/O-stack B and from there it is transported to the desired destination:

```
COPY(E,B); TRANSPORT(B,X);
```

A list of the commands used by the reduction machine editor is given in [HOMMES 79].

6.3. Updating expressions

Updating expressions in an expression-oriented editor means: replace a subexpression by another subexpression. This is always done in the same way:

1. The user enters an expression into the EXPRESSION-field and positions the cursor to the expression in the FA-field which he wants to replace.
2. Via algorithm SCROLLDOWN the expression which is going to be replaced is brought to the top of stack E.
3. Via algorithm INPUT the new expression is generated from the old expression, the program library, the auxiliary stacks, and the input specified by the user.
4. The expression to be replaced is deleted.
5. The new expression is moved from stack B to stack E. A scroll-up operation is performed to return to the previous FA, which now includes the replaced subexpression.

Figure 16 shows the contents of the stacks during the different phases of replacement:

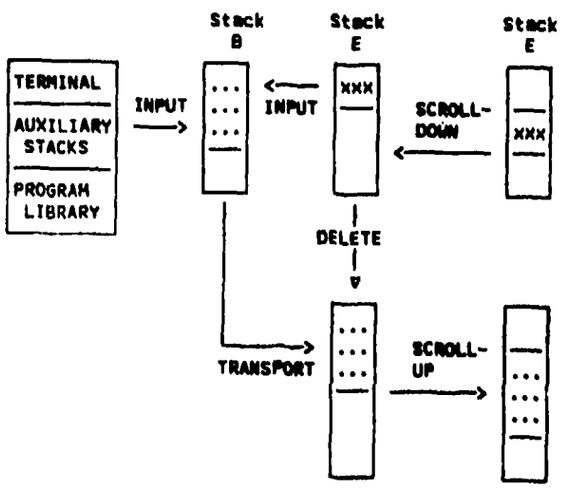


Fig. 16: Contents of stacks when replacing an expression.

Outline of procedure REPLACE:

```
PROCEDURE REPLACE(ADDR: EXPRESSION-ADDRESS);
BEGIN
  SCROLLDOWN(REVERSE(ADDR));
  INPUT;
  DELETE(E);
  TRANSPORT(B,A);
  TRANSPORT(A,E);
  SCROLLUP(ADDR);
END
```

The algorithm INPUT performs the following operations:

1. The expression specified by the user in the expression -field is translated from its external representation to the associated internal representation by using the I/O-table.

2. EMPTY-expressions are inserted for missing subexpressions, i.e. the expression entered by the user is automatically completed.

3. References to other expressions are resolved.

4. When INPUT terminates, a complete expression has been generated within stack B.

There are three references to other expressions which may be used when constructing new expressions:

expression-address:

An expression-address is replaced by its corresponding expression, i.e. the expression-address 1.nil may be used to refer to the first subexpression of the expression which is going to be replaced.

Example: Entering 1.nil will replace an expression by its first subexpression

name of an auxiliary stack or of an expression:

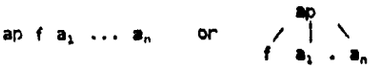
The name is replaced by a copy of the expression which is either on top of an auxiliary stack or in the expression library. This reference is used to retrieve expressions which are moved to an auxiliary stack or to the library by using the copy-command.

Note: Expression references are resolved by applying the basic COPY-algorithm.

This chapter has shown the basic features of an expression-oriented editing-system; [HOMMES 79] gives more information and shows especially how the user can construct programs in such an expression-oriented system.

7. Evaluation of Programs

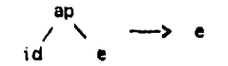
The editing system described so far works for arbitrary languages based on a constructor syntax. Now we are going to restrict this class of languages to applicative languages. These are languages in which an application of a function is always denoted by an expression having the following format:



i.e. a special constructor called applicator followed by a function and its arguments.

Programs written in an applicative language are executed by resolving applications, i.e. by applying functions to its arguments, which is done according to a set of rewriting rules. A rewriting rule specifies the expression by which an application is to be replaced.

Example: The rewriting rule for the identity function is given by



An algorithm which resolves applications can be based on a TRANSPORT-algorithm. The idea is to move an expression from stack E to stack A, but to stop the transport when the following situation occurs: the applicator is on top of stack M, the function is on top of stack A and the arguments are on top of stack E. Then the application is resolved according to the rewrite rule, i.e. the applicator is popped out of stack M, the function is removed from stack A, and the arguments on top of stack E are replaced by the result.

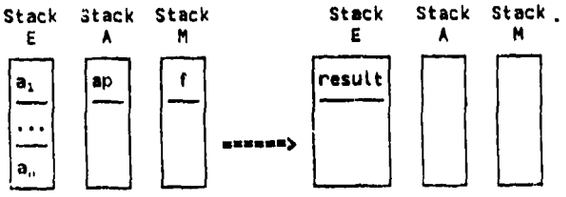


Fig. 17: Resolving an application

Having resolved the application the TRANSPORT algorithm is activated again. When the expression has been moved to stack A all applications have been resolved. The algorithm TRANSPORT(A,E) moves the expression back to stack E.

The editor can be easily extended to allow interactive execution of expressions or subexpressions. Introducing the editor-command E (=Evaluate) into the environment described in II.6.2. will result in:

```
SCROLLDOWN(REVERSE(CEADDR);
EVALUATE;
TRANSPORT(A,E);
SCROLLUP(CEADDR);
```

By using the cursor any subexpression may be selected for evaluation.

References

[BACKUS 72] Backus, J.
Reduction Languages and Variable-free Programming
IBM Research Report RJ 1010, April 7, 1972

[BACKUS 73] Backus, J.
Programming Language Semantics and Closed Applicative Languages
IBM Research Report RJ 1245, July 5, 1973

[BACKUS 78] Backus, J.
Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs
CACM 21, No.8, Aug. 1978, pp.613-641

[BERKLING 76] Berkling, K.J.
Reduction Languages for Reduction Machines
Interner Bericht ISF-76-8, 1976
GMD, Schloß Birlinghoven, D-5205 St.Augustin 1

[HOMMES 77] Hommes, F.
The Internal Structure of the Reduction Machine
Interner Bericht ISF-77-3, 1977
GMD, Schloß Birlinghoven, D-5205 St.Augustin 1

[HOMMES 79] Hommes, F. & Schlütter, H.
Reduction Machine System User's Guide
GMD, Schloß Birlinghoven, D-5205 St.Augustin 1

[KLUGE 79] Werner E. Kluge
The Architecture of a Reduction Language Machine
Hardware Model
Interner Bericht ISF-79-3, 1979
GMD, Schloß Birlinghoven, D-5205 St.Augustin 1

PARALLEL COMPUTER ARCHITECTURE EMPLOYING FUNCTIONAL PROGRAMMING SYSTEMS

John C. Peterson
William D. Murray

University of Colorado at Denver
Department of Electrical and Computer Engineering

Abstract

By using a functional programming system as a machine language, a highly parallel computer can be constructed. A form of lazy evaluation, using incomplete objects, provides a mechanism for constructing a data flow computer which directly executes programs written using the functional program system in a highly parallel manner. Since a data flow architecture is used, this parallelism is not dependent on any specialized parallel language or compiler. This computer consists of three basic components: a set of processors, a shared memory containing only FP objects, and a queue feeding functions to all processors. The design is modular, allowing an arbitrary number of processors, which need not be identical.

INTRODUCTION

A new approach to data flow computers is suggested by functional programming (FP) systems, as described by Backus [1]. By introducing a form of "lazy evaluation", similar to that used by Friedman and Wise [3] in a computer whose machine language is an FP system, a simple yet powerful data flow computer results.

Unlike other parallel computers, data flow processors [2,4,5,6] obtain parallelism directly from its source: the natural data dependancies between operations in a program. Such computers are not bound to parallel languages or compilers, but are able to introduce parallelism into all programs without need of assistance above the hardware level.

FUNCTIONAL PROGRAMMING SYSTEMS

This section will serve as a refresher on FP systems and as a reference for later discussion of FP systems. Only those aspects of FP systems relevant to computer design will be reviewed. A complete description of the FP system used here can be found in Backus [1].

An FP system is described by five things: a set of primitive functions, a set of functional forms, a set of definitions, and the operation of application. Formal systems for functional programming (FFP systems) use objects to represent FP functions.

An object is either an atom, a sequence whose elements are objects, or $\perp$ ("bottom" or "undefined"). Atoms include numbers and identifiers. FP systems whose sequence constructor is $\perp$ preserving will never allow $\perp$ to be an element of a sequence. Only in an FP system whose sequence constructor is not $\perp$ preserving could the sequence $\langle X,\perp\rangle$ be found. The special atom ϕ is used to denote the empty sequence, which is both an atom and a sequence. Sequences will be represented by enclosing the sequence elements in < and >. The application operation is denoted by a :, so the application of the function f to the object x would be written as f:x.

All functions are applied to a single object. Since all functions have only one argument, it is unnecessary to give names to arguments. Because all programs are composed only of such functions, all variable names are completely eliminated. Functions which would normally require more than one argument are applied to a sequence containing all of the needed arguments. A brief list of the primitive functions to be used follows.

n:x	Where n is an integer. Find the nth element of the sequence x.
tl:x	Remove the first element of the sequence x.
id:x	The identity function. Return x unchanged.
atom:x	Tests if x is an atom. T is returned for true, F for false.
eq:<x,y>	Tests if x and y are equal objects.
null:x	Tests if x is ϕ.
reverse:x	Reverse the elements of the sequence x.
distr:<s,x>	Create a sequence of pairs formed by pairing each element of s with x, $\langle s_i,x\rangle$.
distl:<x,s>	Like distr, except the pairs will have x for the first element, $\langle x,s_i\rangle$.
length:x	Find the length of a sequence.

+:<x,y>	Add x and y. (-, x, and ÷ are similar.)
and:<x,y>	And the booleans x and y. (Or and not functions are similar.)
trans:x	Transpose x, where x is a sequence of sequences identical in length.
apndl:<x,seq>	Append x to the left end of seq.
apndr:<seq,x>	Append x to the right end of seq.
apply:<f,x>	Apply the function f to the object x.

A $\perp$ is produced whenever a function is applied to an improperly formed object, such as applying a selector to an atom or using a sequence in place of a number for an arithmetic operation. All functions are $\perp$ preserving, returning $\perp$ when applied to $\perp$. (But see the discussion later of non $\perp$ preserving functions).

Functional forms are functions which use other functions or objects as parameters. Forms are used to create expressions involving functions The functional forms to be used are:

f∘g:x	Compose f and g. Returns f:(g:x).
$[f_1,\ldots,f_n]$:x	Construct a sequence whose ith element is f_i:x.
(p→f;g):x	If p:x is T return f:x, otherwise if p:x is F then return g:x.
$\bar{y}$:x	Return y, a constant.
/f:x	Insert a binary function into a sequence. /f:$\langle x_1\rangle \equiv x_1$;/f:$\langle x_1,\ldots,x_n\rangle \equiv$ f:$\langle x_1$,/f:$\langle x_2,\ldots,x_n\rangle\rangle$.
αf:$\langle x_1,\ldots,x_n\rangle$	Apply a function to all elements of a sequence.
(while p f):x	While p:x = T apply f to x

The state D contains all functions defined by the user. A function definition associates a name (an atom) with a function. Definitions are denoted by def name ≡ function. All function names must be either defined in D or known by the system as primitive functions or forms. Since D never changes during the execution of a program, the set of functions defined for a particular program is static.

THE INHERENT PARALLELISM IN AN FP SYSTEM

The FP forms which directly imply parallelism are: apply-to-all (α), insert (/), and construction ([...]). Apply-to-all creates a sequence by applying the same function to a variety of objects, while construction creates a sequence by applying a variety of functions to the same object. Within these forms function evaluations can proceed in parallel, due to the absence of side effects.

The insert form computes a single result absorbing each element of a sequence into a dyadic operator. If the operation being inserted is associative (i.e., if f:<A,(f:<B,C>) =f:<(f:<A,B>), C> for all objects A,B and C) then this form can be highly parallel. An associative insert can "tree in" the sequence rather than proceeding serially through the sequence. Associative functions would be recognized before program execution and two different insert forms would be used: insert and insert-associative.

An interesting property of these forms is that if a parallel construction form is implemented, then parallel versions of the insert-associative and apply-to-all forms can be expressed using parallel construction. Assuming that the function is being applied to the pair <function, object> (the result of $[2\circ 1,2]$ in an FFP system), then suitable definitions for the apply-to-all and insert-associative functions are:

```
def APPLYTOALL ≡ null∘2→φ;
                 apndl∘[apply∘[1,1∘2],
                 APPLYTOALL∘[1,tl∘2]]

def INSERTASSOC ≡ eq∘[length∘2,1̄]→1∘2;
                  INSERTASSOC∘REDUCEPAIRS

def REDUCEPAIRS ≡ leq∘[length∘2,1̄]→id;
        [1,apndl∘[apply∘[1,[1∘2,2∘2]],
         2∘REDUCEPAIRS∘[1,tl∘tl∘2]]]
```

The function name leq is used for a less-than-or-equal-to function. For the apply-to-all function, if both the apply and APPLYTOALL arguments to the apndl function are evaluated in parallel, then eventually each application will be running in parallel. In the case of INSERTASSOC, the function REDUCEPAIRS will apply the function being inserted to successive pairs in the sequence, halving the length of the sequence. This will be done in parallel, as with APPLYTOALL. The INSERTASSOC function iteratively calls REDUCEPAIRS, which trees in the sequence one level, until the final result (the top of the tree) is reached.

PARALLELISM IN COMPOSITION

Introducing parallelism into the composition form is more difficult. The nature of composition would seem to prohibit any sort of parallelism due to the inherent data dependency between the functions being composed. If it is required that the data transferred between the functions is an object in the usual sense, then parallelism is in fact impossible. If, however, a function is able to form partial results, then these results can be passed between the functions allowing some degree of overlap. These partial results arise from the ability to decompose (or factor) many functions.

To express partial results incomplete objects will be introduced. An incomplete object is an object containing portions which have yet to be determined, but which eventually will be filled in. The FP system requires only one new "object" to express these incomplete objects, the incomplete atom ω. ω will serve as the fundamental unit of incompleteness, capable of assuming any value on completion. An ω can be thought of as a place-

holder, representing the result of an arbitrary function which has not yet been finished.

Every ω will be associated with a completion function. This completion function will eventually specify a value to be used in place of the ω. Formally, any ω should be identified by its completion function. A more casual notation, in which ω's with different completion functions will be given different subscripts, will be used herein. Of course, there may be many references to the result of a single completion function.

When ω is used as a sequence in an append function, a new sort of incomplete object is created. If apndl:$\langle X,\omega_1\rangle$ is evaluated, the result will be denoted by $\langle X,\Omega_1\rangle$. Ω is called the incomplete subsequence, and is used to indicate a section of a sequence, of arbitrary length, which has not yet been filled in. In this example, ω_1 and Ω_1 have the same completion function, yet the result of the completion function will be installed within a sequence in the case of Ω_1. For example, if Ω_1 (and ω_1) complete to $\langle Y,Z\rangle$, the sequence will now be $\langle X,Y,Z\rangle$, not $\langle X,\langle Y,Z\rangle\rangle$. All Ω's will be found within a sequence. Any time that an Ω completes to a non-sequence, an error ($\downarrow$) will result. An Ω is not a separate incomplete atom, but rather a different usage of the basic incomplete atom ω. Any Ω_i will be dependent on some ω_i for its completion function. If ω appears within a sequence, it represents a particular element of the sequence whose value is as yet unknown, but if Ω appears in a sequence, it represents a portion of the sequence itself which is unknown. Any sequence containing Ω will be termed an incomplete sequence; any object containing either ω or Ω will be termed an incomplete object.

Conceptually, an incomplete object is a set of objects. This set contains all possible values the incomplete object may assume on completion. For example, ω would be the set of all objects, $\langle\Omega\rangle$ would be the set of all sequences (including ϕ), $\langle\omega_1,\omega_2\rangle$ would be the set of all sequences of length 2, and so on. A partial ordering of incomplete objects can be constructed using the containment relation between their associated sets. An incomplete object, X, is more complete than another incomplete object, Y, if the set of objects associated with X is a proper subset of the set associated with Y. A complete object is one whose set contains only one member, the object itself.

When a function is applied to an incomplete object, four different situations may arise:

1. The object is not sufficiently complete for the function to have any effect. In this case, the function must be deferred until the object becomes more complete.
2. The function can be applied to portions of the object, but must defer applying itself to other sections of the object.
3. The function can be applied to the object, but the result is still incomplete.
4. The function can be applied to the object and the result is a complete object.

A few illustrations of these cases are:

1. +:$\langle 3,\omega_1\rangle$ cannot be evaluated (at this instant).
2. Reverse:$\langle A,B,\Omega_1,D.E\rangle$ = $\langle E,D,$(reverse:$\langle\Omega_1\rangle$), $B,A\rangle$ = $\langle E,D,\Omega_2,B,A\rangle$, where a new ω_2 has been created to hold the result of (reverse:$\langle\Omega_1\rangle$).
3. 3:$\langle A,B,\omega_1\rangle$ = ω_1. Trans:$\langle\langle\omega_1,\omega_2\rangle$, $\langle\omega_3,\omega_4\rangle\rangle$ = $\langle\langle\omega_1,\omega_3\rangle,\langle\omega_2,\omega_4\rangle\rangle$.
4. 3:$\langle\omega_1,B,C\rangle$ = C. Length:$\langle\omega_1,\omega_2\rangle$ = 2.

A rather subtle problem has arisen here. By postponing the completion of a sequence, the $\lfloor$ preserving nature of the sequence constructor has been lost. For example, if 1:$\langle A,\omega_1\rangle$ is evaluated to A, this result becomes incorrect if ω_1 is completed by $\lfloor$ and the sequence constructor is $\lfloor$ preserving. Thus, it is natural for an FP system which uses incomplete objects to have a sequence constructor which is not $\lfloor$ preserving, preventing entire sequences from being later replaced by $\lfloor$.

(To further allow parallelism, it would be possible to produce other functions which are not $\lfloor$ preserving. An example of such a function would be the and function. If and is defined so that its result is F (false) if either element of the pair it is applied to is F, then and:$\langle F,\omega_1\rangle$ could be immediately evaluated to F.)

Incomplete objects are closely related to the suspensions produced in lazy evaluation [3]. One difference is that incomplete objects imply concurrent function evaluation while suspensions imply delayed function evaluation. Another is that conceptually, incomplete objects stay within the realm of objects (with only ω added), while suspensions are used transparently. The real advantage in using incomplete objects rather than suspensions lies in the clean notation of incomplete objects and the ability to stay within the set of objects.

THE DESIGN OF AN FP COMPUTER

The design goals of the FP computer will be:

1. The computer will use an FP system as a machine language.
2. The memory will be used only for FP objects.
3. The computer will be data-driven; parallelism will result naturally from data dependencies.
4. The computer will be modular, allowing great expansion without any change in the basic architecture.

Goal 1 provides a computer which will enforce a disciplined use of the memory at the hardware level, preventing destructive updating and side effects. Goal 2 allows the memory to be homogeneous. Since only objects are being stored, the memory is not forced into the conventional word and address structure. Goal 3 attempts to produce an ideal data flow computer by putting the burden of parallelism onto the hardware. Goal 4 states that the design should be expandable, allowing great increases in computing power without changing the underlying architecture.

Incomplete objects will be used to produce the necessary parallelism. Two basic principles will govern the use of incomplete objects. First, all functions will be completion functions. This associates each function with a place (an

incomplete atom) for its result. The second principle is that incomplete atoms will be generated by the function apply. This includes the use of apply in most functional forms. For example, f:g:x would be treated as f:(g:x), so that two incomplete atoms would be used, one for the result of g:x and the other for the result of f:(g:x).

The FP computer will have three basic components: A set of processors, a memory, and a READY queue. The processors apply functions to objects, the memory holds these objects, and the READY queue feeds functions to the processors.

The READY queue functions as a "shared program counter". All functions evaluated by the processors must flow through the READY queue. Whenever a function is ready to be executed, it is placed into the READY queque. A queque element (instruction) has four components. The format of a queue element is:

$$\langle \text{function, object, } \omega_{\text{result}}, D \rangle$$

The function and object describe an application to be performed, ω_{result} indicates the atom being completed, and D is the state of the program. D will be constant for all queue elements of a single program. In a multiple program environment, different programs could be distinguished by their different D s.

The memory contains only objects. Objects include queue elements, D's, functions, and incomplete atoms. The memory must be managed, allowing new objects to be created and removing objects which have become garbage. When an incomplete atom is identified as garbage, its completion function must be terminated. Since it is important to remove these garbage functions as soon as possible, garbage should be identified immediately when produced.

All incomplete atoms will have an attached queue, similar to the READY queue. These queues will contain functions which are blocked by an input which is not sufficiently complete. Whenever a function cannot evaluate, it attaches itself to an incomplete atom blocking it. When an incomplete atom is completed (actually, it still can be replaced by an incomplete object, but it will always become more complete), its queue is attached to the READY queue.

The processors take queue elements from the READY queue and execute them. Figure 1 gives a simplified flowchart of processor operation. Three distinct paths exist through this flowchart: one for garbage functions, one for functions blocked by incomplete objects, and one for functions which are executed. Processors have three sorts of functions to deal with: built in functions, defined functions, and forms. Built in functions have some standard representation recognized by the processors; defined functions are fetched from the state, D; and forms are handled through the metacomposition rule. All inter-processor communication is handled by the READY queue and memory. No special inter-processor communication hardware is required. Also, no processor has any state saved between instructions.

MULTIPLE PROCESSOR TYPES

The architecture can be expanded to accommodate different types of processors. The only addition needed is a READY queue for each processor type. When a queue element is ready for execution, it is placed into the READY queue corresponding to the function within the queue element. This allows a system to use a smaller number of processors for functions which are costly to implement or infrequently used. Also, a high speed arithmetic processor would not be tied up executing non-arithmetic functions.

One very useful processor type would be a processor which only checks for executable functions (functions whose object is sufficiently complete to allow execution of the function). This very simple processor would remove this burden from processors with computing abilities.

COMPARISON WITH OTHER DATA FLOW COMPUTERS

A broad definition of a data flow processor [6] is one in which the execution sequence is controlled by data dependencies. Many data flow computers require that a model for the partial ordering of the execution sequence be constructed before execution, at a time when data dependencies cannot be completely located. The FP computer, however, needs no such model since data dependencies are manifested during program execution. Furthermore, the FP computer allows specialized processors and program control is not directed from a single master processor.

The use of a FP system for a machine language induces single assignment behavior [5], which is also found in pure LISP [3,4]. FP systems provide a more practical machine language than LISP [3,4] since FP systems do not use a changing environment or variable names. Selectors are much more suitable for accessing values at the machine level than names.

The placing of a queue element into the READY queue corresponds to firing [2] but the FP computer does not know if the queue element is actually ready for execution. An FP operation may "fire" several times, each time waiting for a more complete input, until the operation is finally performed.

The overhead involved with parallelism lies in encountering functions which are found to be unexecutable due to an insufficiently complete object. This overhead is usually limited for a particular function, since only a limited number of stages of completion are possible for objects. For example, the + function normally will see a maximum of only 3 stages of completion of its argument, such as $\langle \omega_1, \omega_2 \rangle, \langle \omega_1, n_2 \rangle, \langle n_1, n_2 \rangle$.

IMPLEMENTATION OF THE FP COMPUTER

This section outlines those features of the FP computer which relate to parallel processing.

The Functional Forms

Composition: Composition uses an incomplete atom to link the functions being composed. When $\langle f \circ g, x, \omega_{\text{result}}, D \rangle$ is executed, a new incomplete atom, ω_{temp}, is created. The function g is started

by placing <g,x,ω_{temp},D> in the READY queue. The function f is placed in the queue attached to ω_{temp}, in the form of the queue element <f,ω_{temp}, ω_{result},D>. As soon as g produces its first partial result, f will attempt to proceed.

Construction: Construction forms a sequence of incomplete atoms. When <[f_1, ... ,f_n],x, ω_{result},D> is executed, a result ,<ω_1, ... ,ω_n>, is immediately formed. Also, for each f_i the queue element <f_i,x,ω_i,D> is added to the READY queue.

Apply-to-all: The only difference between construction and apply-to-all is that apply-to-all may be applied to an incomplete sequence. If <αf,<x_1, ... ,Ω_i, ... ,x_n>,ω_{result},D> is executed, the result will be <ω_1, ... ,$\Omega_{i'}$, ... ω_n>. The queue element <αf,<Ω_i>,$\omega_{i'}$,D> will be attached to the queue of ω_i. Otherwise, all other functions will be attached to the READY queue as with the construction form.

Insert-associative: When applied to a complete sequence, insert-associative can be implemented in terms of other forms. When applied to an incomplete sequence, this form is similar to apply-to-all.

Condition: There are two ways to implement the conditional form, (p→f;g) : parallel and non-parallel. Both would have the same semantics, but a parallel conditional would evaluate p, f, and g in parallel. This is not always desirable, since considerable processing might be wasted evaluating the alternative which will not be chosen. This is a real problem in loops closed by a conditional, since a parallel condition form would look ahead beyond the end of the loop. Other times, however, parallel evaluation of p, f, and g will speed up execution.

For the non-parallel condition, evaluation <(p→f;g),x,ω_{result},D> will create a new functional form, choose. <(choose f g x), ω_{temp},ω_{result},D> will be placed on the queue of ω_{temp} and <p,x,ω_{temp},D> will be placed on the READY queue. Once p returns a value, the choose form will be activated, which will select either f:x or g:x as a result.

The parallel conditional can be expressed in terms of construction and a new primitive function, cond. A parallel (p→f;g) would be expressed by cond∘[p,f,g], where cond behaves like (1→2,3). The parallelism results from the parallel function evaluation used by construction. When p returns a value, the unused function, f or g, will become garbage and terminate.

Primitive Functions: Different primitive functions require various degrees of completeness before being executed. A few examples are:

+ requires a complete object.

length requires a complete sequence.

3 requires a sequence whose first 3 elements are not Ω's.

id permits any incomplete object.

The only other aspect of primitive functions related to parallelism is the ability of some functions to decompose themselves when applied to incomplete sequences (see "reverse").

Processor Synchronization

Only two operations require synchronization of the processors. First, requests for new objects must be synchronized. This can be accomplished by various techniques, depending on the exact memory organization. The simplest would use a conventional free list protected from multiple accesses with a semaphore. An "intelligent memory" might be able to handle multiple memory requests internally.

The other need for synchronization lies in the only object which can be updated: the incomplete atom. The time between finding an incomplete atom and attaching an element to its queue must be protected from completion of the atom. This could be accomplished with a semaphore on each incomplete atom. Since these queues are not as active as the READY queue and the time duration between finding an incomplete atom and using its queue is short, little time would be lost on processor synchronization.

The READY Queue

The READY queue must be an extremely fast queue, since all functions must pass through it. As long as all instructions put into the READY queue are eventually given to processors, it is not important to force specific queue behavior on the READY queue. Also, it is not necessary to have multiple READY queues for different processors if processors pull only the type of functions they need from a single READY queue, although this could involve unnecessary waiting for the proper function type.

A PROGRAMMING EXAMPLE

A characteristic example of the parallelism introduced by the FP computer is found in a sorting program. A merge-sort program written in an FP system might be:

```
def SORT   = (/MERGE)∘(α[id])

def MERGE  = null∘1→2;null∘2→1;
             GREATER∘[1∘1,1∘2]→apndl∘[1∘2,MERGE∘
                    [1,tl∘2]];
             apndl∘[1∘1,MERGE∘[tl∘1,2]]
```

Since MERGE is associative, /MERGE can be implemented with an insert-associative form. One kind of parallelism will result from the use of the insert-associative: the MERGE function will be arranged in a tree and all merges in a level of the tree will execute in parallel. Another kind of parallelism arises when the MERGE operations produce partial results through the use of incomplete sequences. Each time a MERGE produces an element of its result, this element is immediately fed into the next higher MERGE. A diagram of the data flow is given in Figure 2.

This parallelism was achieved completely by the computer; no explicit parallelism was embedded in the program. This example should serve as an indication of the amount of parallelism which

would naturally occur when a program is run on an FP computer.

CONCLUSIONS

Functional programming systems provide a basis for a computer architecture which introduces parallelism at the most basic level: the machine language. Through the use of incomplete objects, a completely data-driven computer has been designed. Parallelism has been achieved without complex synchronization mechanisms or complex inter-processor communication networks. Furthermore, the computer could accommodate very large numbers of processors for the introduction of a very high degree of parallelism.

This computer has the additional benefit of a structured machine language with simple and clean semantics. No instructions are provided for the introduction of parallelism; this comes automatically. Thus, all programs run on this computer take advantage of available parallelism without the aid of special parallel languages or compilers. Parallelism does not change the semantics of a program, allowing the programs to be analyzed without regard to parallel behavior.

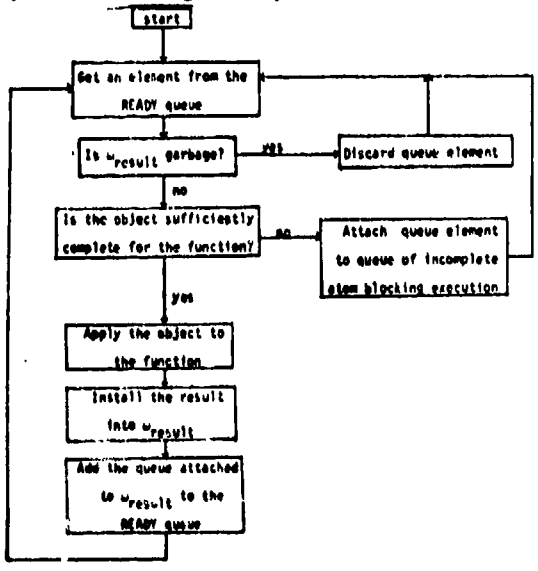


Figure 1

Flowchart of Processor Operation

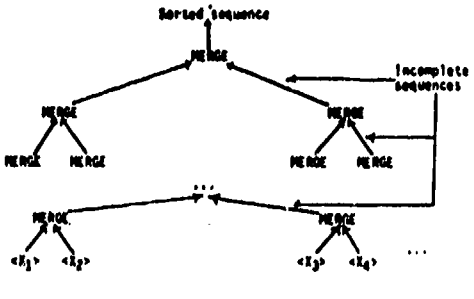


Figure 2

Data flow in a MERGE-SORT

BIBLIOGRAPHY

1. Backus, J.W., "Can Programming be Liberated from the von Neuman Style?, A Functional Style and Its Algebra of Programs," Communications of the ACM, vol. 21, no. 8, August 1978, pp. 613-641.

2. Davis, A.L., "The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine," Proceedings of the 5th Annual Symposium on Computer Architecture, April 1978, pp. 210-215.

3. Friedman, D.P., and D.S. Wise, "Aspects of Applicative Programming for Parallel Processing," IEEE Transactions on Computers, vol. C-27, no. 4, April 1978, pp. 289-296.

4. Misunas, D.P., "Report on the Second Workshop on Data Flow Computer and Program Organization," Report #MIT/LCS/TM-136, Massachusetts Institute of Technology, Laboratory for Computer Science, June 1979, pp. 11-12, 21.

5. Plas, A. et al., "LAU System Architecture: A Parallel Data Driven Processor Based on Single Assignment," Proceedings of the 1976 International Conference on Parallel Processing, August 1976, pp. 293-302.

6. Rumbaug, J.E., "A Data Flow Multiprocessor," Proceedings of the 1975 Sagmore Computer Conference on Parallel Processing, August 1975, pp. 220-223.

On Architectures for Document Preparation

Martin Freeman &
Leon S. Levy

Bell Laboratories
Whippany, New Jersey

ABSTRACT - We claim that the principal limitation in the performance of current document preparation programs lies in the inability of the underlying architecture to efficiently execute the most frequently performed operation -- the movement of data and its reorganization in the computation of line images of the output. We present the design of a unit intended to expedite this data rearrangement, in the context of a macro-architecture, and show how this unit can be generalized to variety of other processing tasks.

1. INTRODUCTION

Architectures are described which utilize VLSI technology to directly address the problems of document formatting to which computers are being applied with increasing frequency in the rapidly evolving field of office automation.

Both a macro-architecture and a micro-architecture are described. The macro architecture presents a framework within which to develop all of the functions associated with document processing. The micro-architecture is a specification of the design of a particular aspect of document processing.

Our particular micro-architecture addresses the area of text formatting. The major component of this architecture is the Fill Line Unit (FLU) which performs a function in DPM's analogous to that performed by the ALU's in conventional machines. It provides a first example of the realization in hardware of the many functions associated with text processing.

2. MOTIVATION

Computer technology is generally described as having progressed through several stages of evolution, usually referred to as generations:

- First generation (1950-1957) - vacuum tubes and miscellaneous main memories
- Second generation (1958-1964) - transistors and random access magnetic core memories
- Third generation (1965-1975) - small scale integrated circuits and random access magnetic core or solid state memories
- Fourth generation (1975-present) - medium scale integration and solid state random access memories

During the same period of time there has been a steady shift from primarily arithmetic and control computation to the mixture of arithmetic and symbolic computation typified by document preparation and the so-called "office automation."

The changes in technology have been reflected in the architecture of the processing units. The introduction of a bus structure was eventuated by the availability of large numbers of registers in the processing unit with the transition to third generation systems. The introduction of cache memories came with the availability of solid-state memories. However, the architecture of computers has not dramatically been affected by the changes in the typical application mix.

In [7] Mukhopadhyay surveys architectural considerations for non-numeric processing and points out that, "With the proliferation of computers in all spheres of human civilization, most of what will be expected of future computers will be non-numerical...Existing computer architecture

does not provide efficient non-numeric computation."

Much of the research in non-numeric processing of late has centered on searching, sorting and pattern matching hardware for database machines[8]. Architectures for document preparation, and in particular text formatting systems, need not use special hardware for searching, sorting or pattern matching.

We believe that the major improvement in computer architecture required by document preparation systems is the rapid and efficient rearrangement of data in memory, with relatively minimal associated processing. The proof of such an assertion is likely to be quite difficult, but the data in Table 1 show the effect of 'line filling' only on an admittedly simple document processor--- roff [11].

# of lines	processing time w. line filling (cpu seconds)	processing time w/o line filling (cpu seconds)
66	.?	.?
163	2.3	1.5
544	9.4	5.3
876	11.4	6.5

Table 1. Comparison of Processing Time W and W/O 'Line Filling'

In Table 1, the same documents were run through the document processor twice, once with 'line filling' in which case lines are right and left justified, and once without 'line filling' in which case the text is printed without rearrangement. In line filling, the text is arranged so that, on each line, the maximum number of words are included and if these do not quite fill the line, then the words are spaced out inserting added blanks between words. Although this incremental processing requires relatively little computation, it is very intensive in data movement.

In this paper we describe a document preparation component, the Fill Line Unit (FLU), which can be used to enhance the capability of machines used heavily for this type of non-numeric computation. The augmentation of architectures by means of such add-on units has many precedents in the evolving architecture of computers: extended arithmetic capability, I/O channels, cache, memory mapping, and direct memory access are such enhancements which have been introduced as the technology became appropriate.

3. MACRO-ARCHITECTURE

We now describe a macro-architecture (see Figure 1) as a framework for explicating the concept of the FLU. In this architecture user text is kept in a Line Memory (LM), a buffer's worth of lines for each active user. The state of a formatting process is kept at any time in a register bank indexed by user. Among the registers are the file descriptor register (FDR), the line address register (LAR) which points to the next line in a user's buffer, a memory data register (MDR) which contains a line fetched from line memory or gotten from an I/O device, and the line count register (LCR) which contains the number of lines left to process in a user's buffer.

Typically, a user's process index is placed in the Bank Select Register causing the user's process registers to be selected. The LAR is used to address the next line in the Line Memory to be processed. This line is accessed and concatenated with the present contents of the MDR, the FLU unit is activated and the result placed back in the MDR. If all the lines of a given user's buffer area have been processed, then a new buffer's worth is brought into the LM.

4. DESIGN AND IMPLEMENTATION OF THE MACRO-ARCHITECTURE

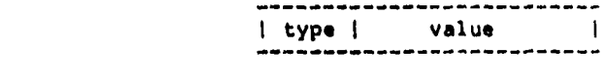
So far we have specified a framework within which a FLU could be utilized. Now we specify the details associated with a text formatting application.

The registers in the register bank have only thus far been partially specified. The text formatter registers in the register bank consist of the left margin register (LMR) which contains the position of the left margin on a line of output text, the right margin register (RMR) which contains the position of the right margin on a line of output text, the page number

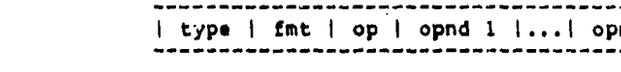
register (PNR) which contains the number of the current page, the line space register (LSR) which contains the number of spaces between output lines, the page length register (PLR) which contains the number of lines in an output page, the header register (HR) which contains the header line to be placed on each output page, the footer register (FR) which contains the footer line to be placed on each output page, the piece register (PR) which contains the piece of the MDR that is left after the leftmost portion of the MDR is output, the header bit (H) which is set if a header is to be output, the footer bit (F) which is set if a footer is to be output, and the fill bit (FL) which is set if the line filling operation associated with the MDR is to be activated.

The text formatter accepts text to be formatted along with commands describing the output format of the text. Ideally, we envision a command language that resembles the language used to edit manuscripts. To be brief, we will confine our command language to be rather conventional (see Table 2). It is essentially identical to that proposed in [3].

Both commands and data are resident in line memory. Lines are organized in terms of flytes (short for flagged bytes) --- there are N flytes to a line. The format of a flyte is



In our case there are two types of flytes---data and commands. For the data flyte the value is the internal data character representation. For command flytes the value is an instruction to be performed, the total format of which is

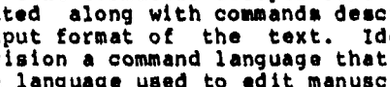


Here the command may be comprised of several flytes. The fmt , format field, describes the composition of the rest of the instruction. For instance, it might specify the number of operands. The op field provides the command which is to be performed. A comprehensive treatment on the selection of instruction formats can be found in [4,5].

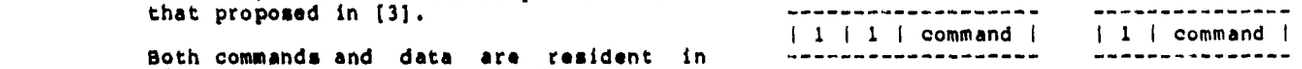
Let us consider an instantiation of this format for our instruction set.

Here we consider a flyte to be eight bits long and character data to be in ASCII representation. Since internal ASCII involves only 7 bits, we can have a type field of 1 bit and still fit a data character in. There will be two command formats---no operand and one operand, distinguished by the setting of the second bit. The no operand format will take up the remaining 6 bits of the flyte, the one operand format will also latch on to the next 8 bit flyte (which must have left bit set) for its argument (range 0-127).

no operand:



one operand:



5. THE FILL LINE UNIT: THE MICRO-ARCHITECTURE

The unit which we have chosen to call the Fill Line Unit, (FLU), plays a role in document preparation analogous to that played by the arithmetic logic unit, (ALU), in scientific computation. Like ALU's, FLU's have a decomposition theory which allows descriptions as serial, series-parallel, or parallel realizations with the appropriate equipment/speed tradeoffs and function of two operands and a carry. In FLU's, the corresponding situation is seen in Figure 2, which is a simplification of the FLU.

Here I is a register which stores i flytes, where i is chosen large enough to generate a complete line image of characters. The data in I at cycle n are used to generate the output line L, and any extra flytes are then stored in O. Thus the functional dependencies are:

$$L_n = g(I_n)$$
$$O_n = h(I_n)$$

Further the new value of I is determined

by the values of O and MDR:

$$In = f(MDRn, On-1)$$

(The analogy between an ALU and a FLU can now be seen more clearly since O is like a carry and L is like a sum.) The two units of combinational logic shown in Figure 2, S1 and S2 are then the primary objects of interest in the synthesis of the FLU.

The above discussion has ignored signals which originate in S2 and set global state information, and signals feeding the global state information into the FLU.

The complexity of the FLU is thus seen, in Figure 2, to depend on the complexity of the units S1 and S2, the remaining units being conventional registers. The role of the S1 unit is to shift inputs from MDR to the right by the length of the data in the O unit, with the non-empty data in O being transferred directly into the leftmost stages of I. S1 performs a uniform shift of all the elements of MDR; symbolically,

$$MDRn-1,k \dashrightarrow In,k+d$$

where Rx,y is the contents of stage y of a given register R at time x, and d is the amount of shift required. If the size of the MDR is s flytes, and each flyte consists of k bits, then the complexity of S1 will be proportional to k*s*log(dmax), where dmax is the maximum possible shift required. In Figure 3, we show a realization of an S1 unit for dmax = 3, k = 1, and s = 3.

In Figure 3, the binary encoded shift control on the left is via a register q which requires log(dmax) bits of storage to control the shift operation of S1, and for the pth shift control bit, qp , stage i is shifted right qp * pp places; i.e. no right shift if qp = 0, and a right shift of pp if qp = 1.

The S2 is considerably more complicated since it performs decoding of the flytes to interpret the embedded control information and non-uniform shifts. We can imagine the structure of the S2, as a uniform cascade of stages as shown in Figure 4. Figure 4 is a conceptual decomposition of the S2 into a linear cascaded array of identical flyte stages. If Ii contains data, the control unit of stage i will pass the control signals through and generate a shift of the appropriate amount. If Ii contains a control flyte, and is therefore not to be shifted to O, then the control information passed to adjacent units is modified, and Ii would be deleted. It might then be necessary for units to the right of Ii to cause a left shift of their contents to L.

We do not give a complete description of the S2 but describe only the logic needed to generate a filled line, omitting the logic needed for the other commands and functions. Further, we shall describe the processing as done in a single clock cycle. Assuming a maximum line length between 128 and 255, the following bus lines are required (of course, using more clock cycles allows fewer bus lines since lines can be shared among functions):

Function	# of bits	Notation
right margin	8	RM[1-8]
right end of text	8	RE[1-8]
word count	6	CT[1-6]
fill status	1	F
fill shift	5	FS[1-5]
fill parameter	2	FP[1-2]
rightmost space seek	1	S
actual shift	5	SH[1-5]

Starting at the left, the word count is set to zero and passed to the right, being incremented at each space following a non-space. The right margin position, r, is encoded on RM. At position r this information is decoded and passed to the left on S until the first space immediately to the right of a non-space, at position s, and position s is then encoded on RE. The difference between RM and RE is then placed on FS and the value of FS divided by CT is placed on FP.

FSi is the incremental amount of shift required at stages following i to right justify the line, and SHi is the actual shift of stage i. SHi and FSi are computed from SHi-1 and FSi-1 with SH1 = 0. If Ii is not blank SHi = SHi-1 and FSi = FSi-1. If Ii is blank and Ii-1 is not blank then if FSi-1 >= FP then FSi = FSi-1 - FP. (Note that FSi + SHi = FSi; the actual shift at stage i and the added shift required is a constant.)

Current component densities are adequate to contain a fully parallel FLU on a single chip for a maximum line size of 130 characters[6].

6. EXTENSIONS TO OTHER TP FUNCTIONS: ENHANCING THE MICRO-ARCHITECTURE

The FLU described in the previous section has shown how to implement many of the classical functions as described in [3]. Other text processors may choose to add functions to these to produce a Cadillac version text processor. While, for reasons of style, we prefer the simpler text processors - especially in an expository treatment - it is worth considering briefly how the architecture described is adaptable to some of these deluxe features. The two which we shall describe are text macros and hyphenation.

6.1 TEXT MACROS

A text macro is a sequence of flytes which replace a single flyte in the source text prior to execution. We shall assume, for simplicity, that the replacement text is fully expanded although, in principle, it need not be. Let m be the macro variable flyte invoking the macro and assume that m occurs in a source line x m y. Assume further that M is the expansion of m. Then after macro substitution the source text is x M y, where M is a sequence of flytes. The transformation from x m y to x M y does not affect x and involves shifting y to the right by length(M) - length(m). Then the substitution text, M, must be placed in the resultant gap. Now the FLU architecture is designed to facilitate exactly this kind of data movement.

Within the context of the FLU, the macro definitions could be stored in an associative ROM. Upon invocation of the macro the replacement text would be retrieved from the ROM and shifted to the appropriate position (using the S1 unit).

Conditional expansion of macros based upon macro variable flytes and external variables (e.g. register contents, transformations on register contents) is also possible. A condition PLA having inputs of macro variable flytes and external variables can generate an output c depending upon which conditions are met. The associative ROM holding the macro definitions would be accessed by the key (m,c) where m is the macro variable flyte. The input (m,c) would act as a composite key for the macro definition.

6.2 HYPHENATION

Most hyphenation schemes depend on some simplified algorithm to approximate correct hyphenation. We shall assume that we have available a small hyphenation box, H, whose function is as follows: Given a sequence of n letters representing the tail of a word (possibly the whole word), and a parameter q, H will determine the place closest to and less than q where a hyphen can be placed. While we have not studied hyphenation algorithms in detail, we do not think that the design of such a unit is extremely difficult.

Now the FLU will gate the word to be hyphenated to H with parameter q indicating where the hyphenation is needed and will use the returned signals to control shifting and line filling.

7. EXTENSION TO THE HOST ARCHITECTURE: ENHANCING THE MACRO-ARCHITECTURE

In Section 3 we provided a strictly vanilla architecture as a vehicle for presenting the FLU. We believe that such an architecture can be generalized to one of a document preparation machine. Pertinent ideas to this end will now be presented, but in the context of a text processing environment.

The architecture of a computer system should be responsive to the needs of the user. In a text-formatting environment, there is a need for entering information from interactive terminals and outputing formatted information from printers or terminals.

Users input requests and the system translates them into actions that it can execute. These actions can be realized by functional units, micro-coded subroutines, etc. For instance, the request format(file descriptor) might be translated by the system into actions which include: transform(line), get(buffer), output(line). Here transform(line) would get the next line from a main memory buffer and format it for printing, output(line) would give the line to a suitable output device, and get(buffer) would replenish the line buffer.

For a given request, its associated actions are related by rules for their application. These rules can be represented by a state diagram where the

states represent the actions and the transitions represent their outcomes (see Figure 5).

In the high-level architecture for the text-formatting machine there exists a supervisory unit which contains the state diagrams for all requests and that sequences through these actions as the requests progress.

Figure 6(a) gives a conceptual view of the Supervisor, Figure 6(b) gives a suitable refinement, and Figure 6(c) gives the execution cycle for the refinement. Note that terminals put requests on a queue which is eventually processed by the Supervisor. The outcomes of each action execution are feedback to the Supervisor for further processing according to the state diagram associated with the executing request.

This supervisory model forms the basis for a multiuser interactive system. (More about this approach can be found in [1,2]). Here the actions associated with the executing request of one user can be overlapped with the actions associated with the executing requests of the other users. Thus we have a pipeline organization where we are always executing different parts of separate requests in parallel.

Expanding on this structure yields a machine architecture as pictured in Figure 7. Users enter requests to create, edit and process text to be formatted. Output can appear on either the initiating terminal or on a line printer.

The Supervisor controls the sequence of action executions while the functional units realize the actions in terms of micro-orders, register transfers, etc. As an example, let us specify in micro-orders the semantics of the transform action:

```
transform(line) =

        Bank Select <- get_queue();
        if ( [LCR] = 0)
                outcome(EXHAUSTED);
        else {
                MDR <- MDR o LM[LAR];
                MDR <- TRAN(MDR);
                LAR <- LAR + 1;
                LCR <- LCR - 1;
                outcome(OKAY);
        }
```

Here outcome(code) places a return code on the Request Queue, get_queue() gets the next entry from the transform action queue, Bank Select is a register which indexes the appropriate user's register set, the operation 'o' concatenates the contents of the next line in line memory to the MDR, and TRAN[MDR] provides the combinational logic function to do the line filling and manipulating operations.

8. GENERALIZATIONS: OTHER APPLICATIONS OF THE ARCHITECTURE

There are several essential features in the design of the FLU which suggest generalizations to functions other than document preparation. First, the FLU operates on a unit of data which is much larger than the elemental storage component typically processed at the instruction level of the computer. This can be considered the outer loop of the FLU control. Second, within the unit of data being processed by the FLU there is a functional pattern suggesting iterative decompositions -- which can be parallel or series-parallel -- which are amenable to replication at the component level. Third, within the data unit processed by the FLU there is a combination of data and control elements similar to a tagged architecture.

The general action of the FLU may thus be understood at the outer level of control as:

```
while (FOREVER) {
        if (DATA UNIT NOT COMPLETE)
                FETCH MORE INPUT;
        else
                PROCESS THE DATA UNIT;
}
```

which in the specific document preparation component case becomes:

```
while (FOREVER) {
        if (OUTPUT LINE NOT COMPLETE)
                FETCH ANOTHER INPUT LINE;
        else
                GENERATE AN OUTPUT LINE;
}
```

In either case the input is a sequence of flytes in which the data and control are intermixed, and the output is a sequence of data flytes. The relationship between the size of the input quantum, the size of the output quantum, and the intermediate storage within the FLU must be studied to obtain optimal performance.

The same processing loop is applicable to a variety of programs in UNIX which have essentially this overall control structure - such as awk [10], sed [9], and grep [9]. (Awk and sed analyze text line by line, while grep searches lines to detect a pattern.) The FLU can be adapted to a variety of programs by having the cascade control logic of the S2 unit under microprogram control.

9. CONCLUSION

Our architectures provide for a synthesis of very large scale integrated circuit technologies and program structure concepts to respond to the needs of office automation.

The macro-architecture and the micro-architecture which we have described combine to provide a state-of-the-art unit suited to an increasing number of applications.

10. REFERENCES

1. Freeman,M., Jacobs,W.W.,& Levy,L.S., "PERSEUS: An Operating System Machine," Proceedings of the Third USA-Japan Conference, October, 1978.

2. Freeman,M., Jacobs,W.W.,& Levy,L.S., "A Model for the Construction of Operating Systems," Proceedings of the 1978 Johns Hopkins Conference on Information Sciences and Systems, April, 1978.

3. Kernighan,B.W. & Plauger,P.J., Software Tools, Addison-Wesley, 1976.

4. Hoevel,L.W. & Flynn,M.J., The Structure of Directly Executed Languages: A New Theory of Interpretive System Design, Digital Systems Laboratory Technical Report 130, Stanford University, March 1977.

5. Flynn,M.J. & Freeman,M., Some Notes on a DEL Basis for Language-Oriented Operating Systems, Computer Systems Laboratory Technical Note 169, Stanford University, November, 1979.

6. Noyce,R.N., "Hardware Prospects and Limitations", in The Computer Age, eds. Dertouzos,M.L. & Moses,J., MIT Press, 1979.

7. Mukhopadhyay,A., "Hardware Algorithms for Nonnumeric Computation," IEEE Transactions on Computers, June, 1979.

8. Langdon,G.G.(editor), "Special Issue on Database Machines," IEEE Transactions on Computers, June, 1979.

9. Kernighan,B.W., Lesk,M.E.,& Ossanna,J.F., "Document Preparation," The Bell System Technical Journal, July-August, 1978.

10. Johnson,S.C. & Lesk,M.E., "Language Development Tools," The Bell System Technical Journal, July-August, 1978.

11. McIlroy,M.D., The Roff Text Formatter, Computer Center Report MHCC-005, Bell Laboratories, October, 1972.

FILL	Fills output lines by packing as many text words as possible onto an output line.
NOFILL	Stops the fill operation.
BREAK	Forces out a partially filled line temporarily stopping the filling process.
SPACE	Breaks and then produces N blank lines.
LINE	Set the line spacing (single, double, etc.).
BEGIN	Break and skip to the top of page N.
LENGTH	Set current page length to N.
CENTER	Causes the next N lines to be centered.
ULINE	Sets underlining logic so that the following words will be underlined.
SLINE	Resets underlining logic so that the following words will not be underlined.
INDENT	Causes all subsequent output lines to be indented N positions.
RMARGIN	Sets the right margin to N.
TINDENT	Breaks and sets the indentation to position N for the current line.
HEADER	Puts the following text on the header of every page.
FOOTER	Puts the following text on the footer of every page.

TABLE 2. Command Set

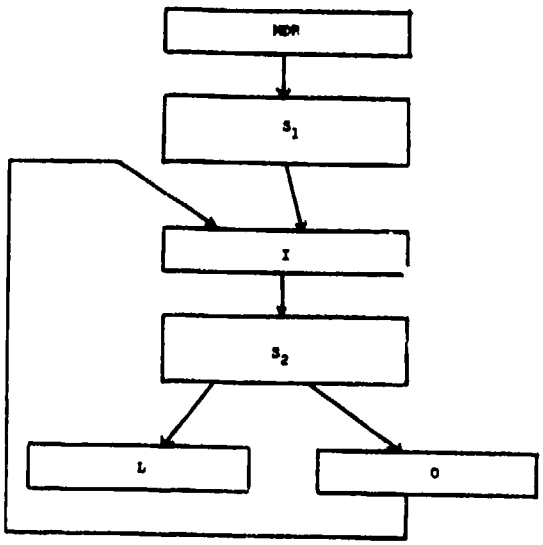


Figure 2. Conceptual Representation of PLU

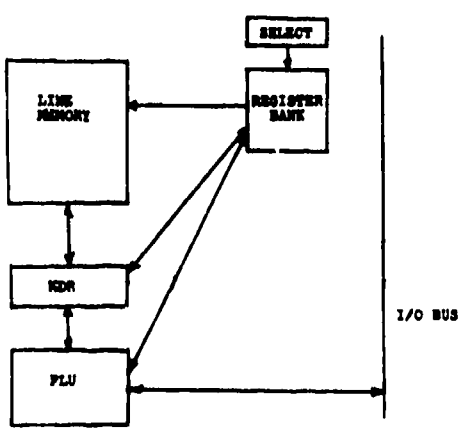


Figure 1. Macro-Architecture

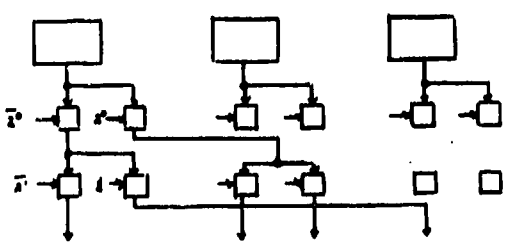


Figure 3. Shift Control Unit R_1
(Interconnections Shown for Least Significant Bit Only)

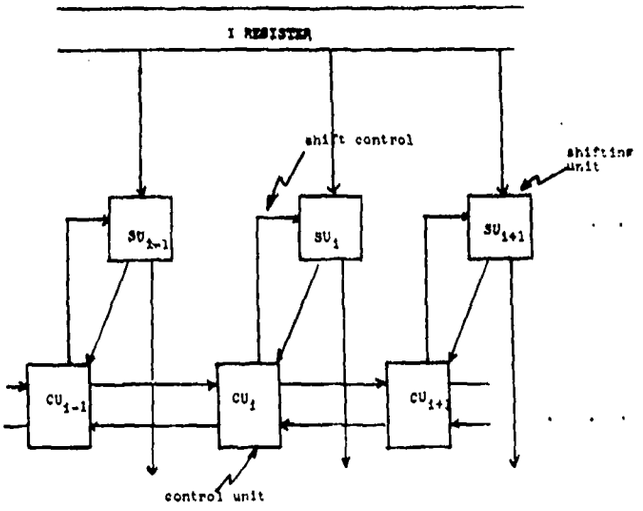
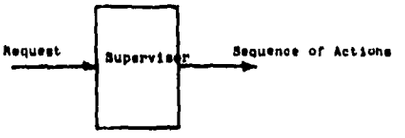


Figure 4. Cascade Decomposition of S_2



(a) Conceptual View

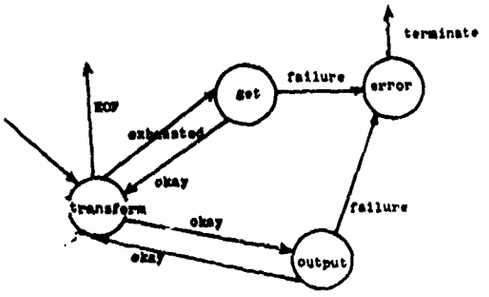
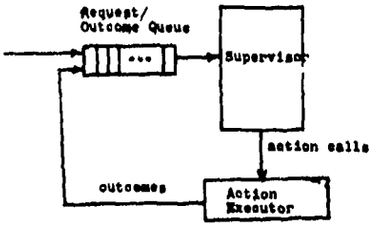


Figure 5. State Diagram for Request format(file descriptor)



(b) A Realization

```
Supervisor() {
    while (queue not empty){
        get next element from queue;
        if (element is request)
            issue first action of request;
        else if (element is outcome)
            if (outcome = FINISHED)
                issue next action of request;
    }
```

(c) Execution Cycle

Figure 6. The Supervisor

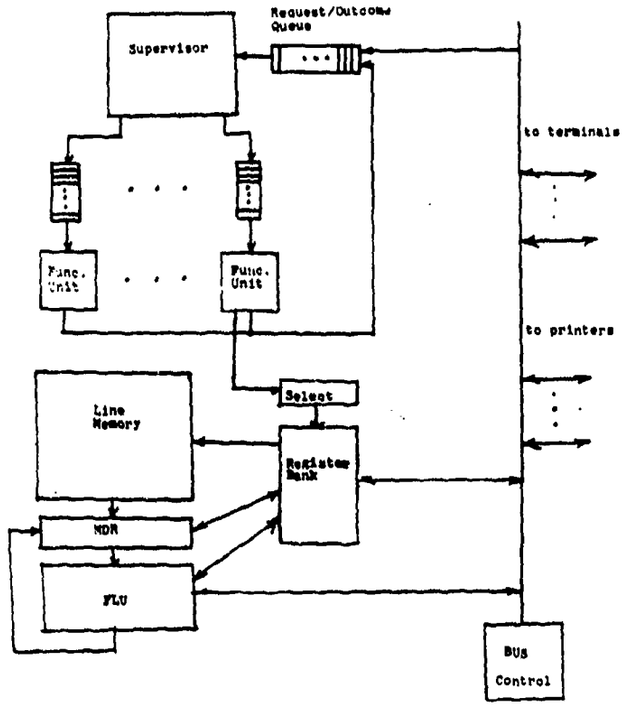


Figure 7. Architecture

A HIGH LEVEL ARCHITECTURE FOR A TEXT SCANNING PROCESSOR

F. J. BURKOWSKI

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF MANITOBA
WINNIPEG, MANITOBA, CANADA

ABSTRACT

This paper discusses the design of a special purpose computer to be used in the scanning of text. The design of this machine allows it to operate at a reasonably high level when performing text searches. This capability not only simplifies the requirements of the translation process used to derive machine code from user enquiries but also enhances the speed of the device which is an essential feature if data is to be scanned while being taken from a rotating storage medium. Of special interest is the design of the term-detection unit which incorporates features which should be of use in a direct-execution arhcitecture, specifically those modules which are responsible for the recognition of keywords and tokens in a stream of source text.

INTRODUCTION

In the past few years we have seen a growing involvement with systems which have as their main function the scanning of extremely large data bases of textual information containing perhaps billions of characters. Examples of such applications include text retrieval systems for intelligence reports, treatises and corpora in law libraries, medical bibliographic services, and large repositories of newspaper articles.

This literature searching is mainly characterized by the fact that the textual information is not structured. Due to the way the information is collected and because of the nature of the information it is usually difficult to provide adequate cost-effective indexing systems. Consequently, if there is any subdivision of the information content, it will be such that the information is grouped into categories which are very extensive in scope. In such a situation, the literature search is accomplished by scanning the entire text. Information is extracted when it satisfies the requirements of a user query which should specify a sufficient number of constraints on the search to produce the required documents and little else.

The internal formatting of the text may be rather inconvenient and limited to standard punctuation although special characters may be used to delimit and hence define various text groupings such as sentences, paragraphs, sections, documents etc.

Various papers [1,2,3,4,5,6] have discussed a variety of architectures for text retrieval and in [7], Hollaar discusses the problems associated with such endeavours and presents a survey of some of the architectures which are of current interest. In [8] Chu suggests that research should explore the hardware, software trade-offs for particular applications involving high-level constructs. This paper is essentially an attempt to bring some of the high efficiency and high performance aspects of direct-execution architecture to the special purpose application of text scanning.

SYSTEM FUNCTIONS

In text retrieval systems, a three step process is involved in the capture of textual information:

1) query translation
2) term detection
3) query resolution

The user terminal (see fig. 1) passes to the system an information request which is expressed as a query. Examples of such an inquisition are as follows:

A — Keyword Search
Retrieve any document that contains the character string A.

(A,B,C,D)#n — Threshold 'OR'
Retrieve any document that contains at least n of the different character strings A,B,C,D. Note that if n=1

this is an "OR" operation; hence the retrieved document contains one or more of the strings A,B,C, or D. If n equals the number of entries in the list, then this is an "AND" operation; the retrieved document must contain all the indicated strings in any order.

A AND NOT B — Logical Expressions
Retrieves any document that contains the character string A but not the character string B.

<A,B>#n — Directed Proximity
Retrieve any document that contains the character string A followed by the character string B within n characters.

[A,B]#n — Undirected Proximity
Retrieves any document that contains character strings A and B within n characters of each other.

A!!!B — "Don't Care" Characters
Retrieves the document with the character string A followed by three arbitrary characters followed by the character string B.

In the next step, the query translator will create the necessary machine code and will send it (along with the required data items) to the query resolution module which guides the behavior of the control unit in the term detector and gathers responses from the term detector in order to resolve queries.

Since it is necessary to scan a vast amount of text, a high speed of execution in the term detector and query resolution modules is of utmost importance. In this design, the scan operations are designed to function at a reasonably high level. During most of the time a search operation will be carried out as the execution of one instruction in the search control unit. If the input text currently being examined contains characters that produce a successful match with a given term, then the execution of various instructions may be effected in order to accomplish some aspect of the query resolution, but in most circumstances the microcode executed during a scan instruction will rapidly skip over text characters which do not match with any of the given terms. As we shall see, it is possible to design hardware facilities which will accomplish some of the query resolution without resorting to the execution of code in the Query Resolution Processor (QRP).

The modular structure of the nucleus of the text scanning system is presented in fig. 2. Because of its functional capabilities, it includes the term detection unit of fig. 1 and, in addition to this, it also involves some aspects of the query resolution block.

The term detection module receives text from a suitable source and attempts to match character substrings in this text with the character string terms stored in the string memory contained within the module.

When a successful match is detected, the match line is given an active signal and the memory address of the matching string is passed down to the status FIFO so that, if necessary, the match can be "logged" for future use by the QRP. The address is also passed to the Interrupt Generation Unit which can be used to implement the "threshold-or" function mentioned earlier. The IGU also decides whether the address is to be logged in the status FIFO.

The delimiter detection unit issues interrupts whenever a delimiter passes in the text stream. It is mainly used to detect the beginning of successive documents in the source text since many of the queries will be related to the contents of a document.

Thus, an interrupt can be initiated for any one of the following events:

a) Detection of a delimiter
b) Detection of a term
c) Completion of a threshold-or during passage of a document.

In all cases, an interrupt line causes the QRP to acknowledge an event which is important to the resolution of a query. If it cannot immediately deal with such an event, all pertinent information is temporarily logged as status in the FIFO buffer until the QRP can find the time to accept it.

TERM DETECTION

The input to the term detector is taken from a source, for example, a disk drive, which can issue a serial stream of characters. It is anticipated that the amount of processing time required between character shifts will be less than 400 nanoseconds. Since typical transfer rates for a disk are about one byte per microsecond this system should be able to accept data directly from a disk without the need for buffer memories or FIFO's.

The heart of the term detector consists of a lengthy shift register which shifts in source text one byte (a single character) each time a shift operation is issued by search control. The shift register is capable of holding 32 characters which are available from the "parallel-out" lines of the shift register. These 32 characters can be compared with any one of 256 strings (or terms) in a "string memory" which has a data bus capable of dealing with 32 characters in parallel. Comparisons are accomplished by a

linear array of comparitors placed between the string memory and the shift register. It is anticipated that each character position in the string memory will involve a 7 bit ASCII code and an additional bit used to signify a "don't care" or unconditional match character.

The string memory is a standard static RAM since the use of associative memory for this function would be very costly at the present time. However, it is obvious that some type of parallel search must be made and consequently, the parallel outputs from the middle four character positions of the shift register lead to an associative memory which also has a word depth of 256. We will refer to these four characters as the "partial match" characters. Prior to the scan operation, the system will ensure that the term in word n of the string memory corresponds to four partial match characters of word n in the associative memory (CAM) (see fig. 3). As text streams through the shift register, a comparison can be effected between the partial match outputs and all the words in the associative memory. If a match is detected, the address of the matching word is derived from an encoder which is driven by the match outputs of the associative memory. This address is fed to the string memory so that another comparison can be accomplished, this time involving the full string. This final full comparison will determine whether the contents of the shift register contain one of the terms required by the user query. With a suitably fast RAM for the string memory, both comparisons can be easily accomplished in the time interval between successive shifts as characters stream off disk.

Our only constraint is that all words in the associative memory be unique. Since most terms in the string memory are not going to be a full 32 characters in length, we should be free to locate a term within its word so that it assumes a position such that the four characters in the partial match positions are different from all the rest.

For example, suppose we are searching the data base for the following ten terms:

" GUYS AND DOLLS "
" THE NIGHT OF THE IGUANA "
" A STREETCAR NAMED DESIRE "
" WHAT MAKES SAMMY RUN? "
" THE DIARY OF ANNE FRANK "
" A LITTLE NIGHT MUSIC "
" SWEET CHARITY "
" THE UNSINKABLE MOLLY BROWN "
" A CHORUS LINE "
" DON'T BOTHER ME, I CAN'T COPE "

Successive words in the string memory might be set up as:

```
|!!!!!!!!!!!!! GUYS AND DOLLS !!!|
|!!!!!!! THE NIGHT OF THE IGUANA |
|!!!!!! A STREETCAR NAMED DESIRE |
|!!!!!!!!! WHAT MAKES SAMMY RUN? |
|!!!!!!! THE DIARY OF ANNE FRANK |
|!!!!!!!!!! A LITTLE NIGHT MUSIC |
|!!!!!!!!!!!!! SWEET CHARITY !!!!|
|!!!! THE UNSINKABLE MOLLY BROWN |
|!!!!!!!!!!!!! A CHORUS LINE !!!!|
|! DON'T BOTHER ME, I CAN'T COPE |
```

while the successive words in the associative memory would be:

```
|GUYS|
|GHT |
|TCAR|
| MAK|
|ARY |
|ITTL|
|SWEE|
|KANL|
|A CH|
| ME,|
```

The ! in the above list represents a don't care or unconditional match character.

As can be seen in the above example each entry in the partial match columns within the associative memory is selected from the corresponding character positions of terms in the string memory. After a parallel comparison of source with all words in the associative memory a successful match will simply indicate a matching substring and the address of the parent term containing that substring. One more comparison with the parent term in the string RAM will serve to verify whether the complete term is in the source text.

It should be noted that in the interest of clarity we have omitted from fig. 3 the additional circuitry required to perform a write operation into the associative memory. Prior to the search, the control unit will define both string memory and associative memory by shifting each query term into an appropriate position within the shift register whereupon a write operation may be executed.

THE INTERRUPT GENERATION UNIT

When the term detector places an active signal on the match line, it is an indication to the rest of the system that the value currently on the address bus is the address of a location in string memory containing a required term. At this time, such an address is accepted by the Interrupt Generation Unit (IGU) and used to aid the processing of a query resolution.

The activity to be initiated by this detection is defined by the contents of a RAM which is established prior to the search. The address value from the term detector is used to access a 6 bit word which is used to control the following two activities:

a) Interrupt enable

If this bit is set, an interrupt signal is issued to the query resolution processor (QRP). The QRP can then act on the presence of the indicated term by executing code associated with the resolution of some particular query.

b) Hardware execution of the "threshold-or"

Another bit in the IGU RAM, the "threshold-or enable", is used to determine whether or not the detection of this term is to be accompanied by the decrementing of a counter which is responsible for the maintenance of the term count associated with a particular threshold-or. The remaining four bits of the word select (via a decoder) one of sixteen counters. Each counter is programmable and can be loaded from the data bus coming from the QRP. Each counter is four bits long, and hence the maximum threshold allowed in such a query is 16.

Since a particular term in any document must decrement the selected counter once and only once, a separate RAM maintains a "hit-list". At the start of a document, all entries in this RAM are set to zero. When a term is first detected (match line high) the presence of a zero from the hit-list and a one from the threshold-or enable bit will cause the selected counter to decrement. This cycle is immediately followed by a cycle which writes a 1 bit into the hit-list and hence any future detection of the term within the same document will not produce an active level on the decode enable line.

In actual practice, it may be necessary to duplicate the hit-list facility since it must be cleared between documents. Consequently, it may be necessary to clear one list while the other is being used.

Finally, it should be noted that a pipeline effect can be incorporated into the design. Once the match address is available, it can be latched for use by the IGU and in this way the activity of the IGU and the processing of the next character in the term detection unit may be overlapped.

CONCLUSION

We have presented a design for a text scanner which uses a term detection unit incorporating random access memory and associative memory in a cost effective manner. An additional module, referred to as the interrupt generation unit, contributes information which greatly enhances the system implementation of high level queries such as the threshold-or.

REFERENCES

1. Hollaar, L. A., "Rotating Memory Processors for the Matching of Complex Textual Patterns," The Fifth Annual Symposium on Computer Architecture, April 1978.

2. Mukhopadhyay, A., "Hardware Algorithms for Non-numeric Computation," The Fifth Annual Symposium on Computer Architecture, April 1978.

3. Roberts, D. C., (ed) "A Computer System for Text Retrieval: Design Concept Development," Report RD-77-10011, Office of Research and Development, Central Intelligence Agency, Washington, D. C., 1977.

4. Roberts, D. C., "A Specialized Computer Architecture for Text Retrieval," Proc. Fourth Non-Numeric Workshop, Syracuse, N. Y., Aug. 1978, pp. 51-59.

5. Stellhorn, W. H., "A Processor for Direct Scanning of Text," presented at the First Non-Numeric Workshop, Dallas, Oct. 1974.

6. Foster, M. J. and Kung, H. T., "Design of Special-Purpose VLSI Chips: Example and Opinions," Technical Report CMU-CS-79-147, Department of Computer Science, Carnegie-Mellon University.

7. Hollaar, L. A., "Text Retrieval Computers," Computer, Vol. 12, No. 3, 1979 pp. 40-50.

8. Chu, Y., "Direct-Execution Computer Architecture," Information Processing 77, IFIP, North-Holland Publishing Co. (1977) pp. 7-12.

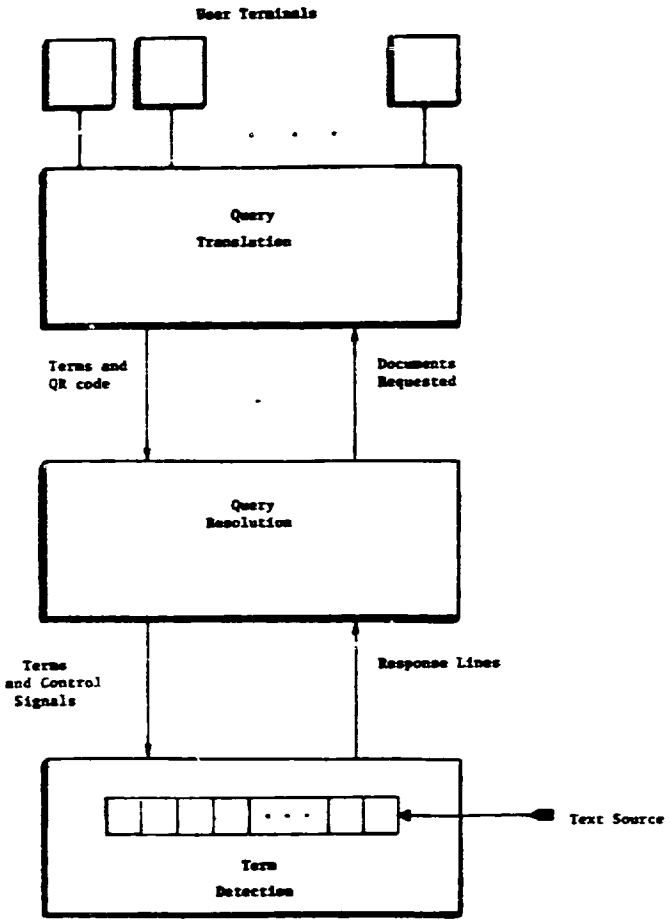


Fig. 1. General Organization of the Text Retrieval System

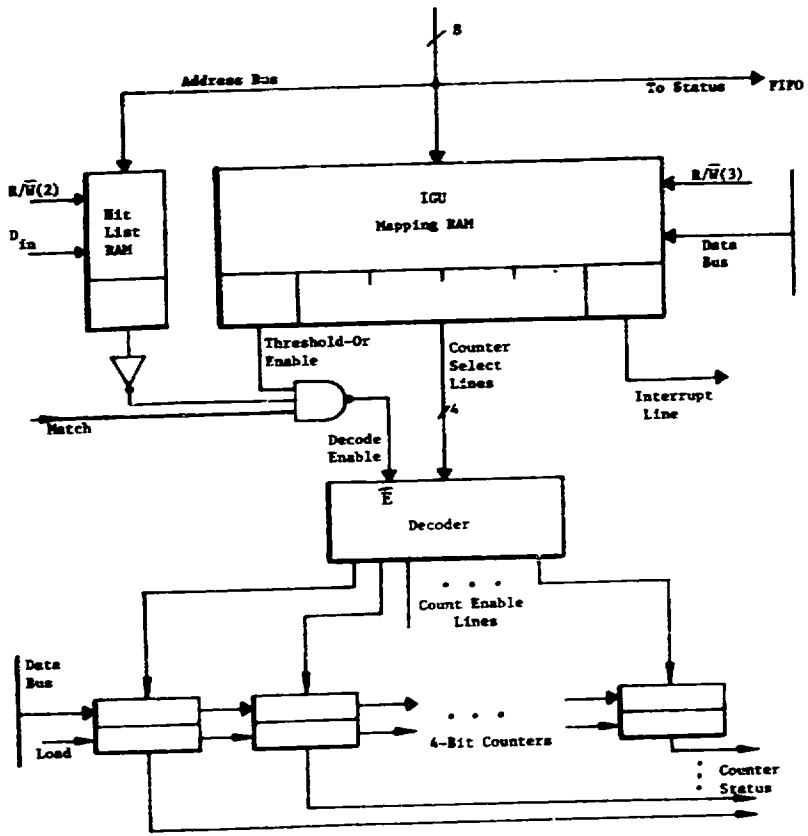
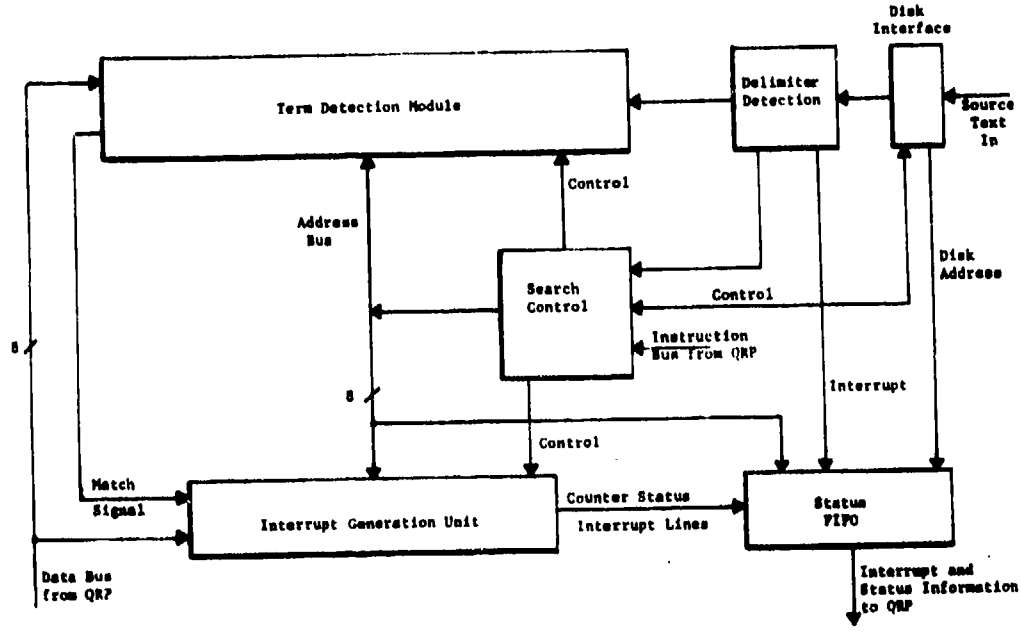


Fig. 4. Interrupt Generation Unit



QRP: Query Resolution Processor

Fig. 2. Text Scanner Organization

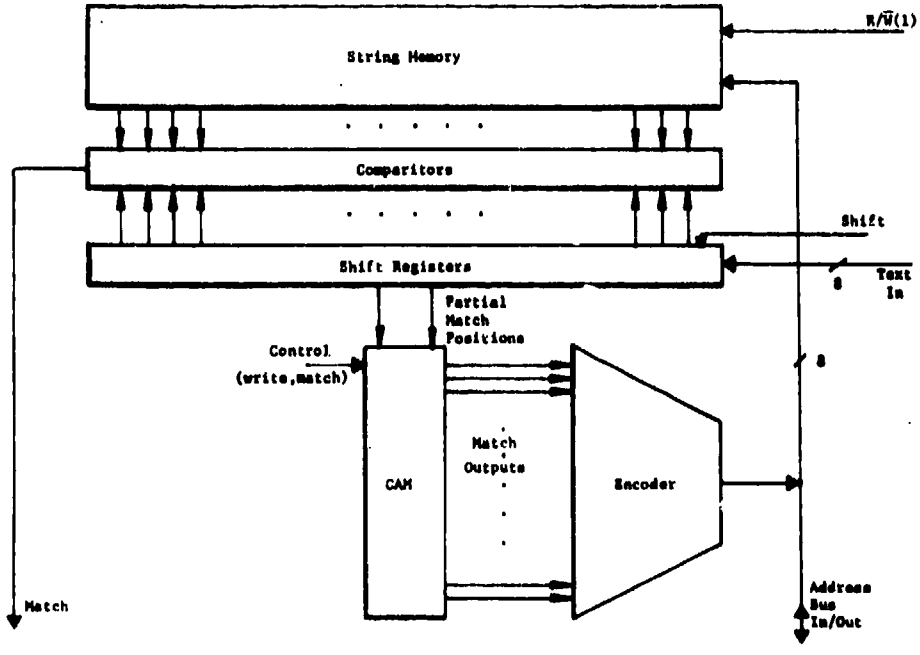


Fig. 3. Term Detection Module

A COBOL MACHINE DESIGN AND EVALUATION

Masahiro YAMAMOTO, Ryosei NAKAZAKI
Minoru YOKOTA, Mamoru UMEMURA

Nippon Electric Co., Ltd., Central Research Laboratories
4-1-1 Miyazaki, Takatsu-ku Kawasaki 213, JAPAN

Abstract

A COBOL machine applicable to an attached processor has been developed. It is characterized by having intensive COBOL machine architecture (COMBAT), highly-specialized hardware structure and compact and efficient host processor interface.

COMBAT architecture has many facilities for efficient COBOL program execution: many internal data, highly functional data descriptors and intensive instructions. COMBAT machine is functionally composed of three processor modules (IFPM, OFPM and EXPM), highly specialized for their functions.

It is found that average COBOL statement execution time is 35% of host processor execution time. A COMBAT machine attains better cost/performance and is useful for a special COBOL processor attached to a medium or large-scale computer.

Introduction

Recent advances in solid-state technology and software crisis due to increased in computer applications are accelerating the research and development of high-level language machines. From the viewpoint of their utilization style, high-level language machines are classified into two categories: a stand-alone processor and an attached processor or an element processor of a distributed-function computer system[1]. Burroughs B1700[2] and NCR COBOL Virtual Machine[3] are typical examples of a stand-alone high-level language machine. PASCAL Microengine[4] from Western Digital Corp. is also a recent interesting product, applied to microcomputer applications.

On the other hand, current marked decreases in the cost of hardware and advent of highly functional processor modules make it not only technically feasible, but economically practical to develop the attached high-level language machine. Taking this trend into consideration, a COBOL machine applicable to an attached processor has been implemented.

In order to attain better cost performance in a high-level language machine, machine architecture and hardware structure design, based on actual user environment are important. For this purpose, an analysis tool is implemented. The analysis tool gathers COBOL user's program profile, including COBOL verbs, operand data attributes and so on. With the help of this tool, a COBOL machine architecture, highly optimized for COBOL program processing, and a COBOL machine hardware structure, greatly specialized for its machine architecture, are obtained.

The COBOL machine can effectively execute major COBOL processing. However, input-output operations, communication control, data base management, software-level virtual memory management and so on, are required for a host processor. Therefore, in a high-level language machine for an attached processor, highly effective, compact and flexible process switching mechanism between an attached processor and a host processor is required. In order to accomplish this function, effective connection interface at the internal bus and firmware level is provided.

COBOL user's programs are translated into highly functional COBOL machine instructions by a software translator, which runs on a host processor.

As an evaluation criterion of high-level language machine architecture, IPF (Instructions Per Function), which indicates how many machine instructions correspond to a source statement, is selected. IPF means machine architecture language proximity. In order to evaluate IPF value and object memory capacity per a COBOL statement, an evaluation tool is implemented.

At present, the COBOL machine is running as a processor attached to a host processor, in which a medium-scale conventional commercial computer (NEAC ACOS series 77 Model 300) is used as a base computer. In the host processor, therefore, FORTRAN, PL/I and COBOL program execution are possible, as well as COBOL program compilation.

As a result of this attachment, COBOL program execution in the host processor is excluded for the COBOL machine. This results in host processor performance enhancement for through-put and turn-around-time.

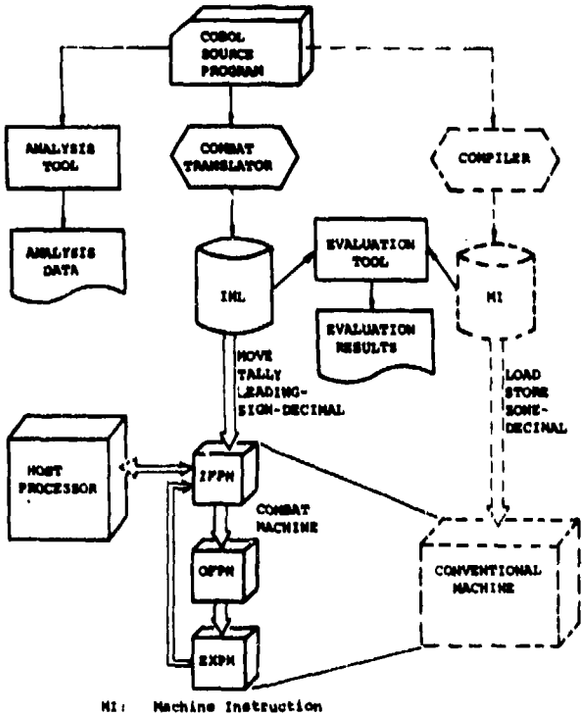
In the following sections, a COBOL machine architecture, COMBAT (COBOL Oriented Machine Basic ArchiTecture), a machine hardware structure, host processor interface and evaluation results are described.

System Overview

Figure 1 shows COMBAT system configuration, including analysis and evaluation tools. The COMBAT system is composed of COMBAT translator and COMBAT machine connected to a host processor.

COBOL programs are translated into highly functional COMBAT instructions by a software COMBAT translator, whose language specification is

compatible with a host processor (ANSI 74 COBOL[5]) for practical use and impartial evaluation of the system. The higher the functional level of a high-level language machine architecture becomes, the simpler a translator becomes. A translator is composed of high-level language dependent part and target machine dependent part. In the COMBAT translator, the processing time and memory capacity for the latter part greatly reduce due to its high functionality.



MI: Machine Instruction
IML: Intermediate Language
IFPM: Instruction Fetch Processor Module
OFPM: Operand Fetch Processor Module
EXPM: Instruction Execute Processor Module

Fig. 1 COMBAT System Configuration and Analysis/Evaluation Tools

Highly functional machine architecture, including host processor interface and intensive hardware structure, closely related to machine architecture, are required for an attached high-level language machine. The COMBAT machine can effectively perform major COBOL functions for data manipulation, table handling, arithmetic operations and conditional operations. Moreover, high performance and compact host processor interface to enable execution of other features are provided to the host processor, e.g. input-output operations, communication control and virtual memory management at the software level. A host processor call instruction in COMBAT machine realizes this function.

COMBAT machine architecture is greatly optimized for COBOL language processing in order to obtain high performance at the machine architecture level. Most COBOL statements, therefore, can be translated into a single COMBAT instruction. Various formats of internal data directly corresponding to all user defined source data are provided.

COMBAT machine has a hardware structure specialized for COMBAT architecture, which is mainly composed of three functionally distributed processor Modules (IFPM: Instruction Fetch Processor Module, OFPM: Operand Fetch Processor Module and EXPM: Instruction Execute Processor Module). Their processor modules are also specialized for their functions using microprogramming techniques and powerful hardware components.

Architecture and Hardware Organization

Architecture

A Cobol Oriented Machine Basic Architecture (COMBAT architecture) has been specified to obtain better trade-offs between hardware and software in high-level language processing. In high-level language machines, it is most significant to decide how much a gap is reduced between a source statement and a machine instruction. In order to attain better performance, the machine instruction set is defined to correspond to a COBOL source statement as closely as possible. Therefore, the following functions are performed during a machine instruction execution.

(i) Data type conversion or adjustment.
(ii) Indexing by index data or subscript data.
(iii) Editing required for data transfer and arithmetic operations.

Machine Instruction Format. Most COBOL source statements are translated into a machine instruction by a software translator, which corresponds to a conventional compiler. A machine instruction is composed of operation code and operand syllables, as shown in Fig. 2. If necessary, a variant syllable or operand number syllable is appended to the operation code. Each operand syllable represents a data item. When the operand is an element in an array, several operand syllables are necessary to specify index or subscript data items.

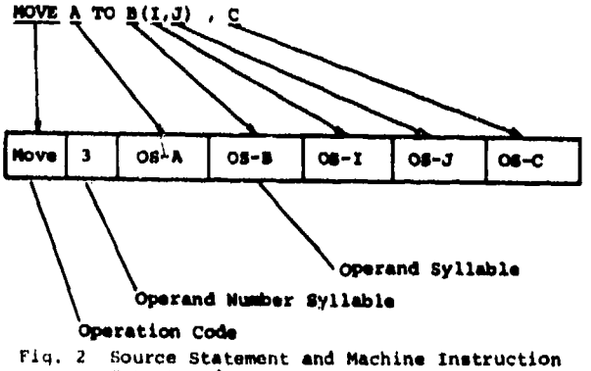


Fig. 2 Source Statement and Machine Instruction Correspondence

Data and Descriptor. COBOL users can handle various data formats in a COBOL program. Since there are only a few data formats directly manipulated in a conventional machine, an object program should convert them into internal formats at run time. This COMBAT machine provides all data formats required in the ANSI 74 COBOL specification. Table 1 lists arithmetic data formats as an example.

Descriptor architecture is adopted to facilitate more complex data description capability for decimal scaling and editing operations. Simple data format operand, however, can be specified without a descriptor to avoid performance disadvantage due to using the data descriptor.

COBOL language allows the user to describe very complex operation in a statement. If it is translated into a single machine instruction, the hardware design becomes too complicated. In the COMBAT machine, complex statements are divided into several basic operations. For example, EXAMINE or INSPECT statement functions are performed with the combination of TALLY and REPLACE instructions. SEARCH and PERFORM statement functions are also translated into several instructions.

Hardware Configuration

The function performed within the COMBAT machine is higher than that for conventional machines. The microprogramming and pipelined architecture is suitable to effectively realize high functionality. In the COMBAT machine, a machine instruction execution is divided into three phases, instruction fetch, operand fetch and execution. Each phase is executed by three independent processor modules, as shown in Fig. 3, Instruction Fetch Processor Module (IFPM), Operand Fetch Processor Module (OFPM), and Instruction Execute Processor Module (EXPM), respectively. These processor modules are connected with each other through First-In-First-Out (FIFO) queue memories. This configuration is intended to be implemented with VLSI chips.

Table 1 Arithmetic Data Formats in the COMBAT Machine

Data Format	COBOL Usage
Signed Binary Short	COMP-1
Signed Binary Long	COMP-2
Singed Packed Decimal	COMP-3
Signed Unpacked Decimal	COMP/DISPLAY (SIGN IS TRAILING)
Unsigned Unpacked Decimal	DISPLAY (NO SIGN)
Leading Signed Unpacked Decimal	DISPLAY (SIGN IS LEADING)
Separate Trailing Signed Unpacked Decimal	DISPLAY (SIGN IS TRAILING SEPARATE)
Separate Leading Signed Unpacked Decimal	DISPLAY (SIGN IS LEADING SEPARATE)

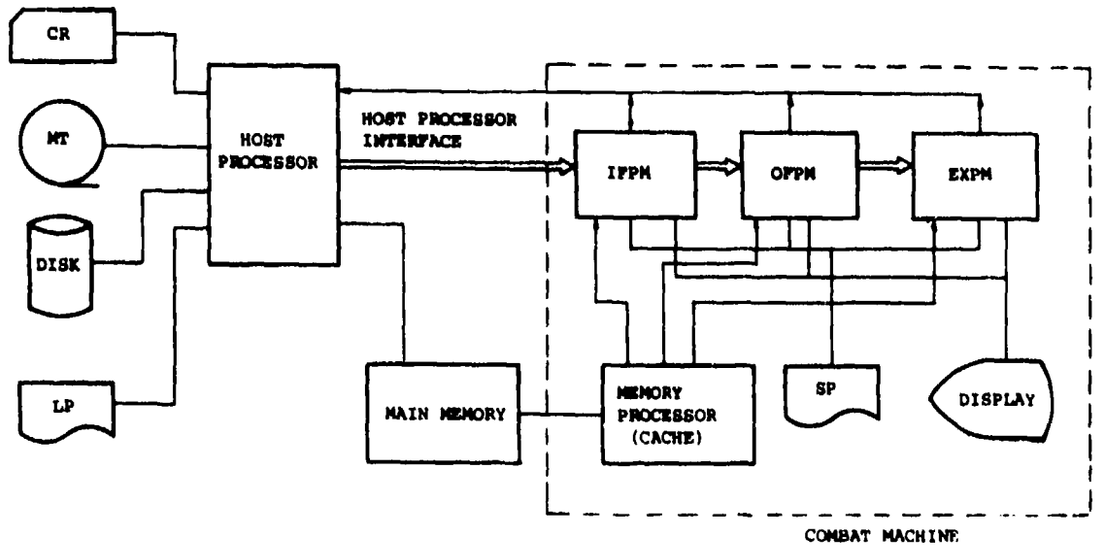


Fig. 3 COMBAT Machine System Configuration

Instruction Fetch Processor Module (IFPM). The main IFPM role is to generate internal forms corresponding to an instruction for easy following manipulations. The operation code and variant syllable are packed into a 32-bit internal machine instruction, as shown in Fig. 4, and transfered to OFPM and EXPM through machine instruction FIFOs. The operand syllables are also packed into a 72-bit internal data descriptor for each operand. Within this process, indexing and subscripting are resolved and an effective operand address is located in the internal data descriptor. Another important role is to control the COBOL program execution sequence. Normally, IFPM continues prefetching according to the sequence represented by such as GOTO, IF and PERFORM statements.

Internal Machine Instruction

Internal Data Descriptor

N: Number of Operands
IN: Instruction Number
ON: Operand Number

Fig. 4 Internal Machine Instruction and Data Descriptor Format

Operand Fetch Processor Module (OFPM). The main OFPM role is to prepare operand data for EXPM, including data fetch, validity check and data format conversion. OFPM fetches data from a main memory. Data contents are examined to validate them. Then, detailed operand attributes are set into an internal data descriptor. For example, it is determined whether data is positive, negative, all space, zero, alphabetic or numeric. When an operand data is used in an arithmetic operation, OFPM converts it into one of two internal data formats, Signed Binary Long or Unsigned Packed Decimal, in order to be easily manipulated in EXPM.

Instruction Execute Processor Module (EXPM). EXPM performs instruction execution as a final stage in a pipeline. To achieve high performance, EXPM installs specially designed hardware units. Especially transfer and editing operations are performed effectively with the aid of these special hardware units, because these operations are most frequently used in COBOL programs.

These processor modules are composed of bipolar bit-slice sequencers (AMD 2900 series). Their instruction cycle time is 200 nsec. IFPM and OFPM micro instruction length is 48 bits and EXPM is 72 bits long. Control storage sizes for IFPM, OFPM and EXPM are 1K, 2K, 3K words, respectively. IFPM and OFPM are implemented with 37 and 42 boards, on which a maximum of 80 ICs can be installed. An EXPM is implemented with 25 boards, which can install a maximum of 200 ICs.

Host Processor Interface

The system is composed of the COMBAT machine and a host processor, as shown in Fig. 3. In this section, the interface between these two processors is described. A COBOL source program must be translated into a COMBAT object program, before the program is processed on the COMBAT machine. The COMBAT machine executes COBOL language processing functions independently from the host processor. The host processor is responsible for this translation and also for miscellaneous functions. For example, I/O statements (OPEN/CLOSE/DISPLAY), inter program control statements (CALL/EXIT PROGRAM) and communication control statements (SEND/RECEIVE) are categorized as such functions. These statements are translated into HOST CALL instructions by the translator.

The COMBAT machine is physically connected to a host processor by a shared main-memory interface and a shared bus interface. Data and program code are accessed by two processors through the shared main-memory interface. Control signals are transfered through the shared bus interface.

Shared Main-Memory Interface

Main-memory can be accessed by both the COMBAT machine and the host processor. Data and programs are located on a host virtual storage space as a unit of segment. Therefore, it is necessary to translate the virtual address into a real memory address, every time a segment is accessed. The COMBAT machine has an address translation mechanism called Memory Processor. For high speed translation, the Memory Processor has 8 pairs of virtual and real address mapping registers in conjunction with an associative memory device. Once a segment is accessed and the address translation has been performed, the address mapping registers contents are effective, as long as the segment stays at a certain real memory location.

When the segments have been relocated by Virtual Memory Manager (VMM: runs on a host processor), the address mapping registers contents must be cleared. Moreover, when the COMBAT machine accesses a segment which is not in the main-memory, the segment must be moved to the main-memory from the secondary storage. The host processor executes this function for the COMBAT machine (VMM CALL).

Shared Bus Interface

Host processor bus is directly connected to the COMBAT machine. To control the bus, a special host machine instruction, named SUPERVISE COMBAT, is provided in the host processor. This instruction is developed into a host micro-code, called COMBAT Support Firmware. Its process flow is shown in Fig. 5. Under the control of the COMBAT Support Firmware, information can be transfered to and from the COMBAT machine through the bus. Therefore, transfering is possible if, and only if, the host processor is executing the SUPERVISE COMBAT instruction.

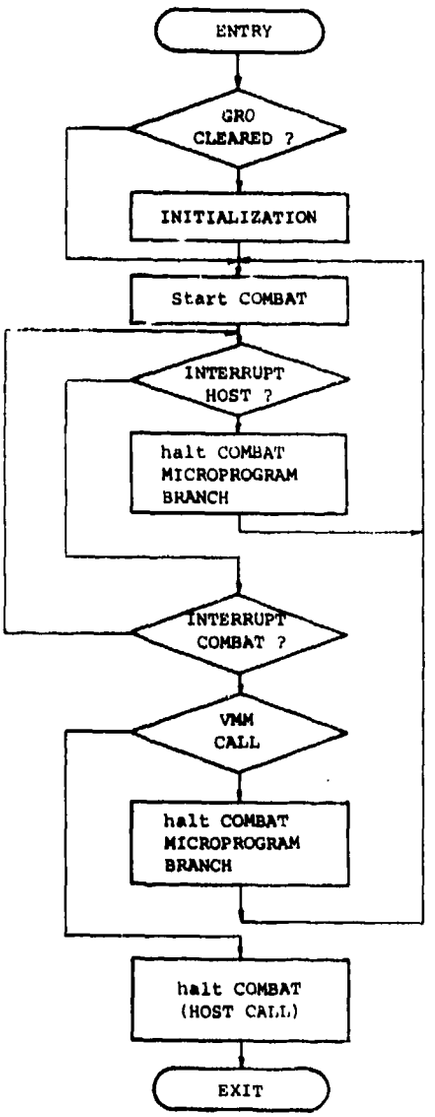


Fig. 5 COMBAT Support Firmware

COMBAT Machine and Host Processor Interaction

A translator program, runs on a host processor, generates four kinds of segments. Three of them are mainly accessed by the COMBAT machine: COMBAT object code, COMBAT descriptor and COMBAT data segment. The other kind is a host object code segment called COMBAT Support Program. It includes the SUPERVISE COMBAT instruction and other codes for the execution of functions to be processed on the host processor mentioned above. The COMBAT Support Program structure is shown in Fig. 6.

At the beginning of a COBOL program process, COMBAT Support Program prepares the execution environment. Segment addresses are loaded in base registers (BRs) and other information in general registers (GRs). Especially, BR4 is set to the top address of the COMBAT code segment, and GR0 contents are cleared. GR0 is used as a flag to control the COMBAT Support Firmware execution.

Then, a host processor executes the SUPERVISE COMBAT instruction, that is, the COMBAT Support Firmware runs. COMBAT Support Firmware generates an initialization signal to the COMBAT machine, and transfers the segments' information through the shared bus interface. The BR4 contents are transfered to the COMBAT machine's instruction counter.

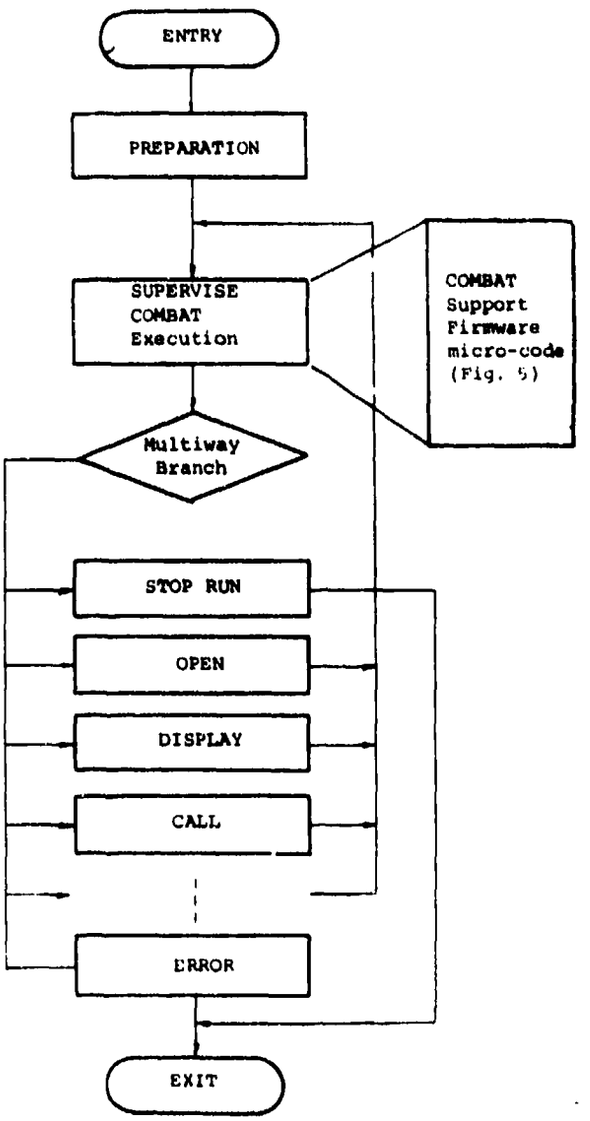


Fig. 6 COMBAT Support Program

Then, the COMBAT machine starts to fetch the COMBAT instructions and descriptors from the segments, prepared for the COMBAT machine, through the shared main-memory interface. After that, COMBAT Support Firmware enters a microprogram loop until an interruption condition occurs, either on the COMBAT machine or on the host processor.

When an interruption condition occurs, the COMBAT Support Firmware halts its supervising process. At this time, the COMBAT machine cannot access a new segment or generate a new HOST CALL instruction to a host processor, until the COMBAT Support Firmware restarts its process. However, other processes inside the COMBAT machine can be executed continuously.

A host interruption causes a microprogram branch to an interruption process part. After interruption process completion, microprogram control returns to the COMBAT Support Firmware and restarts the COMBAT supervising process, or dispatches to another host process. In the latter case, contents for base registers and general registers, related to the COMBAT execution, must be saved. This process is necessary for multi-programming control.

COMBAT machine brings about an interruption in two cases. One is when the COMBAT machine requires access to a segment which is not in the main-memory. In this case, the COMBAT Support Firmware stops its supervising process and calls a host Virtual Memory Manager software routine to move the segment to the main-memory from secondary storage. The other interruption occurs when the COMBAT machine encounters a HOST CALL instruction in the COMBAT code segment. This time, the COMBAT Support Firmware completes its execution, and the COMBAT Support Program takes a host machine cycle.

Next to the SUPERVISE COMBAT instruction in the COMBAT Support Program is an analysis routine for the HOST CALL parameter. The parameter is fetched from the COMBAT code segment through the shared main-memory interface. According to the analysis result, COMBAT Support Program executes one of the functions to be executed by the host processor described before, e.g. EXIT PROGRAM, SEND, DISPLAY, etc. After the HOST CALL instruction execution, the COMBAT Support Program sets the GR0 and executes the SUPERVISE COMBAT instruction again. Detecting that the value in GR0 is not equal to zero, the COMBAT Support Firmware skips the initiation phase and continues its supervising process. If the HOST CALL instruction was a STOP RUN or ERROR instruction, COMBAT Support Program stops its execution.

Evaluation Results

The COMBAT system is evaluated from the aspects of translation from a COBOL program to COMBAT machine instructions and their execution. For this purpose, COMBAT translator and COMBAT machine execution are compared with the host COBOL compiler and host processor instruction execution, respectively. In order to clarify the effect of an attached high-level language machine, an attempt was made to determine how much work load is excluded from the host processor.

As an evaluation measure at the COBOL program level, five typical user programs were chosen. Also, for COBOL statement level evaluation, a COBOL statement mix, consisting of 15 typical COBOL statements, was selected, based on actual user application programs.

Translator Evaluation

In order to clarify the difference between COMBAT and host machine architectures, instructions per function (IPF) have been measured. COMBAT machine and host processor IPFs for the statement mix are 1.7 and 5.5, respectively. These values show remarkable COMBAT architecture proximity to COBOL source statements. The COMBAT architecture brings the following effects on COMBAT translator.

- Translator program memory reduction
- Decrease in translation time
- Object program memory reduction

Improvement degree for these effects is influenced by translation processor unit, machine architecture, translator description language and translator design algorithm. In order to evaluate the difference between COMBAT and host machine architectures, COMBAT translator is composed in the same way as the host compiler, except for the code generation phase. These effects are evaluated with five COBOL user programs, collected from various application areas. Table 2 shows the results of the COMBAT translator performance, compared with the host compiler.

Table 2 COMBAT Translator Performance

Translator Program Capacity	Pre-code Generation	90%
	Code Generation	81%
	Total	94%
Translation Time	Code Generation	66%
	Total	92%
Object Program Capacity		59%

Host compiler = 100%

Both COMBAT translator and the host compiler are divided into pre-code generation part and code generation part. The pre-code generation part design is dependent on the source language and independent of the object machine architecture. On the other hand, the code generation part design depends on the object machine.

Translator Program Capacity. The instructions per function for the COMBAT architecture is markedly reduced. Therefore, the code generation part capacity is 19% less than the host part capacity, in spite of preparing unique functions for COMBAT architecture. The unique functions include generation of data descriptors, multi-operand instructions and host processor codes. The COMBAT translator pre-code generation part memory capacity is almost the same as that for the host compiler, because of their source language dependency and object machine architecture independency. Total memory capacity in COMBAT translator becomes 6% less than the host compiler.

Translation Time. COMBAT translator execution time is measured with software monitor and compared with the host compiler, as shown in Table 2. COMBAT translation time, in code generation part and whole translator, reduce to 66% and 92%, respectively.

Object Program Capacity. COMBAT object program capacity reduces to 59% of the host object program, as shown in Table 2. Object program capacity effects the performance in executing the object program, from the effective memory use aspect. This memory reduction brings about good effect on program locality.

Execution Time Evaluation

Average Statement Execution Time. The average statement execution times in COMBAT machine and the host processor are evaluated. Memory access time and memory usage ratio are also evaluated. These evaluation results are shown in Talbe 3.

Table 3 Execution Performance for Statement Mix

	COMBAT	Host
Average Statement Execution Time	0.35	1.00
Memory Usage Ratio	70%	40%
Memory Access Time	0.80	1.00

COMBAT average statement execution time becomes one third in comparison with the host average statement execution time. The major reasons for this COMBAT machine performance improvement are considered as being:

1) Machine architecture

Highly efficient COMBAT architecture leads to less instruction fetching and data accessing operations, due to compact object code, as shown in Table 2. For example, most literal data are directly described within the instruction and subscripted data address is calculated using a data descriptor. As a result, literal and subscripted data are efficiently accessed.

2) Hardware configuration

Memory access time from each COMBAT processor becomes longer than that from the host processor. In order to improve COMBAT memory access time, a cache memory is provided. Memory access time from the COMBAT machine with the cache memory reduces to 80% of that from the host processor, as shown in Table 3. In spite of this memory access improvement, the COMBAT memory access ratio, 70%, is higher than the host processor memory usage ratio, 40%. This high usage ratio is accomplished due to COMBAT machine pipeline configuration, in which each processor independently generates memory requests and rapidly executes each part of a COMBAT instruction. Rapid execution in each processor is realized by special hardware units, consisting of high speed register files, programmable logic arrays, etc.

COMBAT performance improvement in the COBOL statement mix, in which most statements are very simple, is limited by memory access operation, as shown in the memory usage ratio. Highly functional COMBAT architecture and extensive COMBAT hardware are not sufficiently utilized in this situation. On the other hand, a COMBAT machine has highly efficient machine instructions for complex COBOL statements, 'STRING' and 'INSPECT' and for complex data attribute manipulation, like a subscript and decimal point scaling. Therefore, COMBAT performance improvement becomes larger for these complex statements.

Application Program Execution Time. In order to make a program-level evalution, COMBAT machine execution times for application programs, including input/output and other exclusive operations, are compared with the host processor execution times. In addition, for clarification of effects due to the attached COBOL machine, through-put and turnaround time improvements, for application programs in the host processor, are measured.

Conclusion

A COBOL machine architecture (COMBAT architecture), a COBOL machine hardware structure (COMBAT machine) and several evaluation results have been presented. The COMBAT architecture and COMBAT machine structure are specified to become optimum from both machine architecture and hardware design sides. The COMBAT architecture is highly optimized for COBOL program processing. The COMBAT machine is greatly specialized for the COMBAT architecture.

In addition, the COMBAT machine is aimed to be mainly a COBOL machine, attached to a host processor. Therefore, effective and compact host processor interface is provided.

As a result of architecture optimization and hardware specialization, a highly efficient and low cost COBOL machine was obtained. Moreover, simpler and higher performance software translator than a conventional compiler was attained, due to high COMBAT architecture functionality.

It was found that the COMBAT machine is useful for a special COBOL processor attached to a medium or large-scale commercial computer. In addition, the COMBAT machine is applicable to use as an element processor for a distributed-function computer system.

Acknowledgement

The authors would like to express their grateful thanks to Mr. K. Nezu for his encouragement, and to Mr. K. Hakozaki for his advice. The research reported in this paper was accomplished as a part of a research project on Pattern Information Processing System by the Agency of International Trade and Industry.

References

1. Nilson, R.N., Distributed-Function Computer Architectures, COMPUTER Vol. 7 No. 3, March 1974 pp. 15-16

2. Wilner, W.T., Design of the Burroughs B1700, Proc. AFIPS SJCC Vol. 40, 1972 pp. 489-497

3. Shapiro, M.D., The Criterion COBOL System, Proc. AFIPS NCC Vol. 47, 1978 pp. 1049-1054

4. The MICROENGINE Company, WD/90 Pascal MICRO-ENGINE Reference Manual, 1979

5. American National Standards Institute Inc., American National Standard Programming Language COBOL, X3.23, 1974

A DIRECT HIGH-LEVEL LANGUAGE COMPUTER ARCHITECTURE SCHEME*
(A Computer With a Unified Language Which are the Same Both Inside and Outside—One of Researches on New Architecture of General Purpose Computers)

Gao Qing-Shi

The Computing Technology Institute of The Academy of Sciences of China.

Abstract

This paper is divided into two parts. In first part, using the method of recursive definition, we describe the machine Language and give proper explanations. In second part, we briefly discuss the main parts of implementation of this machine language. We don't attempt to use this machine language for replacement of concrete design of the computers, and only in principle, give a discussion of the units which must be altered to match this machine language. Since time and space are limited, it is only a brief discussion. The first part can be referenced by users and the second part can be referenced while designing machines.

Contents

Introduction

Generally, the languages inside the machines are different from those outside the machines. The internal languages pay more attentions to considering engineering factors, for example, having powerful capability of solving problems, saving devices and so on. The external languages are to consider the facility for use. For instance, ALGOL is used to provide the facility for scientific computation users and COBOL is used to provide

the facility for the commercial users.

Naturally, people can consider, firstly, whether we can slightly modify common machine language (e.g. the language used in the computer 104) to bring the further facility for the users, and secondly, whether we can properly modify common external language (e.g. ALGOL) to execute it directly by computer without adding too many devices.

In this paper we chiefly discuss the second point, and the first point is discussed in the paper "A machine language ridding of the dependency of address."

All of the discussions are preliminary and specialistic and are not for being used directly in the computers. We can imagine that we make a line to link two terminals-One is general machine language and another is ALGOL, the first point being near the first terminal and the second point near the second terminal. For a specified machine, there are a lot of points (i.e. schemes) to be chosen, we must choose it according to concrete conditions. For example, the price of the components is very low, the reliability of the components is very high and there is a associative memory and so on. All of these should be considered as engineering technical conditions. ALGOL can be chosen as a machine language for a special scientific computation machine.

So called computer design, essentially, is to choose schemes based on considering special use requirments and technical conditions. Whether the scheme is good or not depends not only on the rightness of the choice but also on the size of the choice set. In this paper, we discuss three sorts of expressions instead of one sort. For a particular machine, people can arbitrarily choose one sort, two sorts. Or all the three sorts. After we have these three sorts of synthetic schemes, we can make it easy to determine which one we prefer among the seven possible schemes.

---

* This paper was published in CHINA in 1963.

PASC-HLL*

A HIGH-LEVEL-LANGUAGE COMPUTER ARCHITECTURE FOR PASCAL

Jean-Pierre SCHOELLKOPF

Computer architecture
IMAG
BP 53X - 38041 GRENOBLE cédex
France

ABSTRACT

This paper proposes a Tagged Architecture for PASCAL oriented computer architecture. All variables are associated to Variable Descriptors and all data types are described by Type Descriptors. The proposed instruction set is directly defined from HLL statements, ordering the expressions in a Polish form and keeping inside the computer the control structure defined by PASCAL. A hardware computer is next presented which executes the above code by means of five specialized microprogrammed processors working in a pipelined manner.

INDEX TERMS

High Level Language computer architecture, Tagged architecture, Self-identifying data, Polish notation, Pipelined execution.

INTRODUCTION

A first section in this paper presents a new approach for the definition of an instruction set and data representation: it is based on the principles of tagged architecture. The second section briefly presents the currently built PASC-HLL computer that executes the machine language presented in Section 1.

SECTION I - PASC-HLL LANGUAGE DEFINITION

I - INTRODUCTION

Classical machines are working on typeless data considered as a collection of binary digits structured as bytes or words. Except for integers or characters, there is no direct relation between the H.L.L. data types and the hardware types (the ones known by the machine). It is clear that data type definition is the most interesting characteristics of PASCAL language: so it seems to be important to emphasize the problem of "making the hardware suit the language, i.e. to define hardware data types that suit the PASCAL ones. Moreover, we try to define an Instruction Set which suits PASCAL, in the way that it could be the simplest and the most compact executable code compiled from PASCAL, keeping its structured programming feature inside the machine itself.

* Project supported by French contract SESORI n°78-204

A PASCAL programmer is allowed to "define" his own data types (so-called Software types) by structuring basic types (so-called Hardware types). Next he may "declare", inside each procedure, a set of local variables. Finally he writes his program as a structured sequence of PASCAL instructions manipulating his variables. Such a simple description of PASCAL programming directly leads to a simple architecture for a PASCAL oriented computer architecture : its instruction set, so-called PASC-HLL, can be reduced to a manipulation of the programmer-defined variables and the internal operations can be directed by the programmer-defined data types. Such an approach is that followed by K.JENSEN when defining the P.Code (a Pseudo-Code for an hypothetical stack computer [1]). Several implementation of P. Code interpreters are available on mini or microcomputers, but none of them really implements the original P.Code which is based on a TAGGED architecture. Moreover P.Code is rather far from PASCAL for its control instructions: the original PASCAL syntax no more exists in P.Code. So we propose to keep PASCAL control structures in PASC-HLL to simplify debugging and introduce a new kind of software reliability, by providing a computer that"knows" an expresion, an If-Then-Else or control loop structure ; it is a Syntax-oriented architecture [10].

II - THE PASC-HLL DATA STRUCTURES

Using the principles of self-identifying representation, we propose a TAGGED architecture [2] with a basic entity : the Variable Descriptor V.D. associated to each declared variable. When accessing a V.D., the machine must get enough information to perform an operation specified by the programmer. A fixed format was chosen to match with either 16 bits or 32 bits words and to simplify V.D. addressing :

V.D.= (8 - bit TAG,8 - bit STYPE, 16 - bit SVALUE)
TAG = (I Bit, P bit, 2 - bit S, 4 - bit HT).

II.1. Description of the variable descriptor format

a/ V.D. is the basic information referenced by the program : it allows the machine to get all information about the associated variable. Field TAG gives all the hardware description : firstly, the Hardware Type is encoded in 4 - bit HT, indicating one among 16 basic types known by the machine (they are listed in Table 1). If the variable is a PASCAL pointer, then bit P is set. Bit I indicates wether the value of the variable is present in field SV or not ; if

not, field SV holds an address relative to a segment whose number is given by field S, according to Table 2.

b/ Field ST (Software Types) holds an index in the Software Type Table where all the programmer defined types are described. When ST is ZERO, there is no Software Type, so the variable type is the hardware one, for example an 8 - bit integer with hardware bounds (-128, +127). An example of software type descriptor is given in Fig. 1, showing an ARRAY type descriptor holding Lower and Upper Bounds, Element Size, Element TAG and STYPE and finally the Array Size.

All these informations are pointed to by the ST field of an ARRAY variable whose access allows the machine to compute different operations depending on the operator. As an example, Bound Checking, address calculation and building of a Variable Descriptor for the indexed element for the INDEX operator :

```
is LB ≤ Index ≤ UB ?
if yes then : SV  := SV(Index-LB)*STEP
              TAG := Element TAG
              ST  := Element ST
```

c/ Field SV (Software Value) holds either the value of the variable (if it can be represented by 16 bits) or the address of it in the other cases: i.e. for long values, for indirect values necessary before an assignment or for structured types (arrays, records or files). A particular case is that of PASCAL pointers: their value is an address, so bit P is set, and bit I is set or not depending on wether the pointer value is present or not in SV.

II.2. The PASC-HLL stack mechanism

Since PASCAL is a block-structured language, the PASC-HLL machine requires to have a stack mechanism for nesting procedure during execution. If a BASE register is associated to each Lexical Level, it is well-known that the Internal Name of any variable can be built as a couple (Lexical Level, Displacement) = (LL,D), and that this name can be used during execution to access the Variable Descriptors. Previous implementations of that structure are well-known (Burroughs [3], MU-5 [4], etc...). However, it is important to note that parameters must be considered as local variables inside a called procedure, but they must be evaluated in the context of the calling procedure. So we define two steps: firstly a Procedure Variable Descriptor is accessed by a CALL (LL,D) instruction which allows the machine to fetch and store the Formal Parameter Descriptors. Next actual parameters can be evaluated and assigned, after a possible conversion. Finally, another instruction so-called ENTER can check that the correct number of parameters was assigned, next it computes the Mark Stack Control Word [3] and fetches the Local Variable Descriptors before entering the procedure code. Fig.2 describes the stack mechanism.

Such a structure allows a simple and compact addressing mechanism: a positive displacement (00..63) for local variables and a negative one (-64..-3) for parameters is associated to the Lexical Level to form the Variable Internal Name. Fig.3 gives the VIN encoding.

It is now possible to define the Access Instructions whose operand is a variable Name: a 6-bit opcode is associated to a 10-bit Variable Namefor 4 basic instructions: REF and INDD allows the machine to access a Variable Descriptor (with an indirection in the case of INDD), ASSIGN asks the machine to assign a new value to the variable, and CALL allows the access to a Procedure Variable Descriptor. Other miscellaneous access instructions are CLEAR, SET, INCR, DECR to implement frequently used operations on variables such as I: = I+1 or I: = 0 (see table 3). Now, we can show the simple I-PASCAL language.

III - THE PASC-HLL LANGUAGE STRUCTURE

We have just presented access instructions and assignments. Now it is time to say that we choose the PASCAL expression to be translated into POLISH form expressions, and that the PASCAL control instructions will be translated into equivalent PASC-HLL control instructions.

It is clear that the PASC-HLL program will have the same structure as the PASCAL program it has been translated from. An example is given in Fig.4 which shows the equivalence: PASC-HLL offers the same structured programming facilities as PASCAL, needing the machine to base its control structure on the principles of control segments defined by a couple: (Entry Address, Return Address). Inside a control segment, the Program Counter PC is incremented, but syntactic rules must be satisfied: a Polish form expression must be completed before an ASSIGN instruction is fetched and an expression cannot start with an operator, an INDEX operator cannot be applied on any operand: its first operand must be an ARRAY the second one must be a SCALAR.

IV - THE PASC-HLL SEGMENTS

Compiling a PASCAL program generates a PASCAL-HLL Code Segment holding a Types Descriptor Table, a Constant Table and, for each procedure, a couple of two elements: firstly the formal parameters and local variables descriptors, secondly the executable code. An element number is associated to each procedure: it is considered as the "value" of the procedure variable, inside its Variable Descriptor.

During execution, the PASC-HLL machine needs to access to other segments: the first one, so-called Context, holds the nested procedure pointed to by the BASE registers, and the second one, so-called Dynamic, is used for dynamic allocation and accessed by means of "pointers". The Code Segment may be duplicated as an External Code Segment holding "system" or "library" procedures.

So the PASC-HLL machine knows four different segments: that feature allows it to manipulate relative short addresses which can be translated to become absolute addresses. It is then possible to have truely reentrant code and data, and immediat protection between all the segments.

SECTION 2 - PRESENTATION OF THE PASC-HLL COMPUTER

I - INTRODUCTION

After the above presentation of the PASC-HLL language that was defined from the language PASCAL, we propose a hardware computer to execute that machine-language. It is the Pipelined Architecture Slice Computer for High-Level Language, so-called PASC-HLL computer, currently built using AM 2900 family [5].

Pipelined architecture is characterized by a high degree of parallel operations concurrently running inside the computer [6]. Seceral attempts have been made to build high performance pipelined computers [7][8] : their efficiency depends on the way they are programmed and requires a lot of pre-processing for generating optimized Code. The PASC-HLL computer is characterized by a new approach, based on a natural decomposition of the work to be executed,as explained in [9]: "a Pipeline Polish String Computer".

II - PIPELINED EXECUTION IN PASC-HLL

The PASC-HLL order code is syntacticaly in Polish notation, but execution is not made using a Push-down Stack, but a FIFO evaluation queue. That structure makes appear that desynchronisation between instruction fetch, access to Variable Descriptors and execution of operators can be achieved. The first station in pipeline, so-called processor PINS, fetches the next instruction from Main Memory : it executes the Control Functions (Loop control, conditional branch, procedure call and return ...), and sends Access Instructions to an Access station, so-called processor PAC, and Operators to an Operating Station, so-called processor POP. Internal instructions sent by PINS to PAC or POP go through FIFO instruction queues. Variable Descriptors fetched by PAC are sent inside the FIFO evaluation queue managed by another processor, so-called Local Storage Processor LSP.

So, several PASC-HLL instructions are concurrently in different steps of processing, either in Fetch, or Access, or Operation. Moreover, processor LSP manages a FIFO Dependency Queue, which allows to solve the delicate problem of accessing a Value which is not yet modified, but is known to be modified just later. That queue is made up of sixteen 12-bit word Content Addressable Memory that holds the Internal Names of the variables which are to be modified or have just been assigned: it holds the Working Set of the program, achieving good performance for data access and reducing the amount of Memory Accesses. If the Working Set is less than 16 variables (or parameters), no memory access is needed except for assignments: all the Variable Descriptors are inside the CAM. Hardware design of the PASC-HLL computer is now completed: FIVE independent processors, realized using bipolar 4-bit slices, are controlled by FIVE microprograms (the total size is 32 Kbits), with a cycle time equal to 150 nanoseconds.

The PASC-HLL computer is designed to be inserted in a large scale computing center, as a specialized CPU connected to a "Host" computer Main Memory.

The Memory Interface Processor inside PASC-HLL translates virtual addresses sent by the PASC-HLL internal stations (PINS, PAC and POP) into absolute memory addresses according to the Memory allocation made by the "host" system.

The PASC-HLL computer structure is given by Fig.5.

III - THE DEPENDENCY PROBLEM

Suppose, for example, that the high-level instruction "X ← <expression>" has been prepared in both PAC and POP instruction queues, or is in the process of execution by processor POP, when an instruction of access to the variable X enters the access processor PAC. That processor (PAC) must be able to detect the fact that both "X ← <expression>" and "Access X" instructions refer to the same variable X, since the "Access to X" instruction might have to be differred until the completion of the "X←<expression>" instruction, otherwise the "Access X" would not fetch the correct value of that variable, but its old value.

The proposed solution uses a Content Addressable Memory, organized in a FI-FO mode, which holds the names of the variables whose modification is expected (access to their value must be deferred), and the names of the variables which have just been modified.

When an instruction "X←<expression>" is fetched by the access processor PAC, a Dependency Descriptor is pushed into the Dependency Queue. That Descriptor holds the internal name of the variable X (i.e. a Lexical Level and an Offset). From that time, further references to variables are processed through the Content Addressable Dependency Queue, and the name of the variable X is known to "match" with a Dpendency Descriptor. In the same time, processor POP might have completed the execution of the "X←<expression>" instruction. So processor PAC can find either the new value of the variable (just stored by processor POP), or a Dependency Descriptor. The second case is processed as the creation of an Deferred Descriptor. As several deferred accesses to the same variable may occur, all the Deferred Access Descriptors related to that variable are linked together, and they are replaced by the new value of the variable on the completion of the expected assignment instruction.

In the example illustratedby Fig.6, processor POP has just completed the modification of variable C, and it is currently evaluating the expression to be assigned to variable D. In the same time, processor PAC has created a Dependency Descriptor for variable D and it has fetched three "Access D" instructions which has been processed as three Deferred Access Descriptors, since the new value of D is not yet evaluated. After Completion of the modification of variable D, the state of the queue will be as illustrated by Fig.7.

Using the above mechanism, an "Access X" instruction can be deferred until the completion of the "X←<expression>" instruction (assignment). A Deferred Access Descriptor is pushed into the evaluation queue. All the Deferred Access Descriptors are linked together, eliminating a number of memory references equal to the number of linked Descriptors.

IV - THE CONDITIONAL BRANCH PROBLEM

Both PAC and POP processors may be considered as SLAVES of the PINS processor, in that they only execute the internal instructions that they receive from the PINS processor, which is thus considered as the MASTER of the control. However, when a conditional branch occurs, the PINS processor is not able to choose the right next instruction, since the conditional expression is being evaluated at the same time, but it can choose one instruction among all the possible next instructions (generally two). The probability for a WRONG choice strongly depends on the context in which the conditional branch occurs: it is much lower for the end of a loop than for a classical IF-THEN-ELSE statement. Given that the different high-level conditional statements are distinguished by different PASC-HLL instructions, the PINS processor knows the context and can choose the more probable next instruction. When a choice has been made, we say that the PINS processor enters a Conditional State, characterized by the fact that its activity is limited to a "preparation" work. In particular, if a conditional branch is fetched again during this conditional state, no choice is made, but the PINS processor stops ans waits for the resolution of the first conditional branch.

When the value of the conditional expression becomes available in the POP processor, the PINS processor knows whether its choice was wrong or not.

In the case when the choice was right, all processors can go on without any modification. In the other case, all the prepared work must be disabled : this is achieved by emptying the input instruction queues of both PAC and POP processors which hold wrong instructions, and by updating both evaluation and dependency queues in which the sequences of wrong operands or wrong deferred variables must be deleted: this work is achieved by processor LSP.

V - HOW TO SAVE THE EVALUATION CONTEXT

The evaluation context, represented by the intermediate state of the evaluation queue, must be saved when a "function call" occurs within an expression. When the "CALL instruction" is fetched by the PINS processor, a special order is sent to the POP instruction queue. Since several function calls can be nested, a SAVE area is allocated on the top of a push - down stack managed by POP. Then, processor PINS which knows the current state of the evaluation queue, generates a sequence of orders towards the POP instruction queue. Thus, the current state of the evaluation queue is saved by both POP and LSP processors before the function is entered, all previous intermediate results being compacted into the save area.

When the function is returned, processor POP is able to restore values, and the evaluation process goes again.

CONCLUSION

This paper has briefly presented both machine language and computer structure of PASC-HLL computer. It could be necessary to mention that pipelined execution of Polish String was already described in [9] and is not explained again here. That design shows a new kind of machine-language, very compact (up to 4 times more compact than a classical machine-language) and very near the High Level Language bringing a new kind of software reliability during execution. The PASC-HLL pipelined architecture is potentially capable of high performance, since five microinstructions are executed each cycle.

Its global performance is equal to the number of memory accesses which are independently made by three independent processors inside the computer, allowing an optimum use of the Main Memory. Moreover there is a strong relation between HLL programming and hardware processing which works with the programmer-defined variables in the programmer-defined control environment.

REFERENCES

[1] K.JENSEN, "A Pseudo-Code compiler for an hypothetical stack computer", a PASCAL program listing.

[2] E.A.FEUSTEL, "On the advantages of Tagged architecture", IEEE Trans.on Computers, vol.C-22, n°7, july 1973.

[3] Burroughs B5500, Information Processing Systems, Reference Manual, Burroughs Corp., Detroit, Michigan, 1964.

[4] "MU-5 Basic Principles", Manchester University, England.

[5] "The AM 2900 Family data book", Advanced Micro Devices, Sunnyvale, California, 1978.

[6] "System 360 Model 91: machine philosophy and instruction handling", IBM Jr & D, 11, n°1, January 1967.

[7] R.M.TOMASULO, "An efficient algorithm for exploiting multiple arithmetic units", IBM Journal January 1967.

[8] J.E.THORNTON, "Parallel operation in the Control Data 6600", AFIPS FJCC 1964, Washington DC.

[9] G.BAILLE & J.P. SCHOELLKOPF, "A pipeline polish string computer", AFIPS NCC 1976, New York City.

[10] Y.CHU, "Concepts on high-level language computer architecture", University of Maryland, College Park, Maryland.

HT value	Hardware Type		HT value	Hardware Type
0	8-bit integer		8	Character
1	16-bit "		9	Char. String
2	32-bit "		A	32-bit real
3	64-bit "		B	64-bit "
4	8-bit powerset		C	Boolean
5	16-bit "		D	Bool. String
6	32-bit "		E	Array
7	variable length powerset		F	Record

Table 1 : the PASC-HLL Hardware Types

Tag bits

I	P	S	comment
0	0	..	value in SV field
0	1	01	pointer value in SV field
1	0	00	indirect value in CONTEXT
1	0	01	" " in DYNAMIC
1	0	10	" " in MAIN CODE (constant)
1	0	11	" " in EXT. CODE "
1	1	00	indirect pointer value in CONTEXT
1	1	01	" " " in DYNAMIC

Table 2 : Segment Addressing modes

Instruction	meaning
REF(ll,d)	access a Variable Descriptor
INDD(ll,d)	build an Indirect Variable Descriptor
ASSIGN(ll,d)	assign a value to a Variable
CALL(ll,d)	access a Procedure Descriptor
CALF(ll,d)	access a Function Descriptor
PARAM(SP,-n)	assign a value to a Parameter
CLRV(ll,d)	clear a Variable (I:=0)
SETV(ll,d)	set a Variable (B:=true)
INCV(ll,d)	increment a variable (I:=I+1)
DECV(ll,d)	decrement a Variable (I:=I-1)

Table 3 : the PASC-HLL access instructions

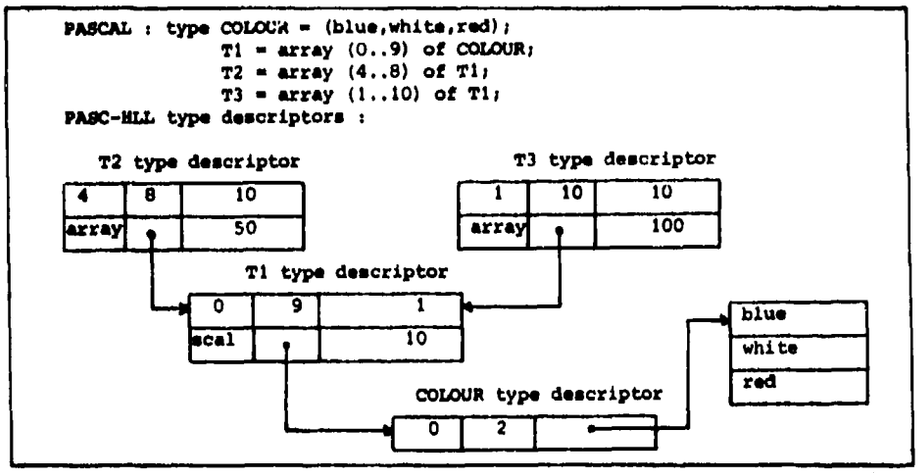


Figure 1 : example of ARRAY type descriptors

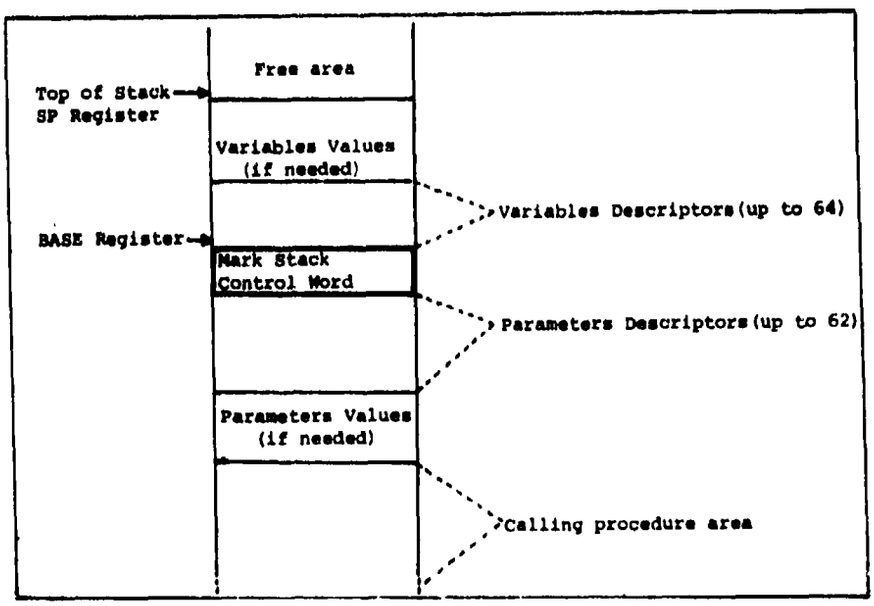


Figure 2 : the PASC-HLL stack mechanism (CONTEXT segment)

Encoded Internal name 9 8 7 6 5 4 3 2 1 0		Comment
0 0	D	D=0..255 for Global Variables
0 1 0	d	d=0..63 for Local Variables
0 1 1	d	
1 0 0	d	
1 0 1	d	d=-64..-2 for Parameters
1 1 0	d	
1 1 1 0	n	Special WITH addressing
1 1 1 1	n	Special SP relative addressing

Fig.3 - The PASC-HLL Variable Internal Name encoding (10-bit)

PASCAL structure	PASC-HLL structure
X := T(I+1)-1 ;	REF(T), REF(I), INC, INDEX, DEC, ASSIGN(X) ;
while exp do stat ;	LOOP(↓),exp,WHILE,stat,ENDLOOP ;↓
for I := exp1 to exp2 do stat;	exp1,exp2,FORUP(↓), ASSIGN(I),stat,ROFUP;↓
case exp of 0,1:stat1; 3,4:stat2; else:stat3 end;	exp,CASE(↓), LIT(0),LIT(1),OF(↓),stat1,FO, ↓LIT(3),LIT(4),OF(↓),stat2,FO, ↓stat3,FO;↓
if exp then statement;	exp,IF(↓),statement,FI;↓
PROCNAME(exp1,exp2);	CALL(PROCNAME),exp1,PARAM(SP,-3), exp2,PARAM(SP,-4),ENTER;
SET := (3,I..J);	LIT(0),LIT(3),ADDELEM, REF(I), REF(J), SUBSET, UNION, ASSIGN(SET);

Fig.4 - Equivalence between PASCAL and PASC-HLL structures

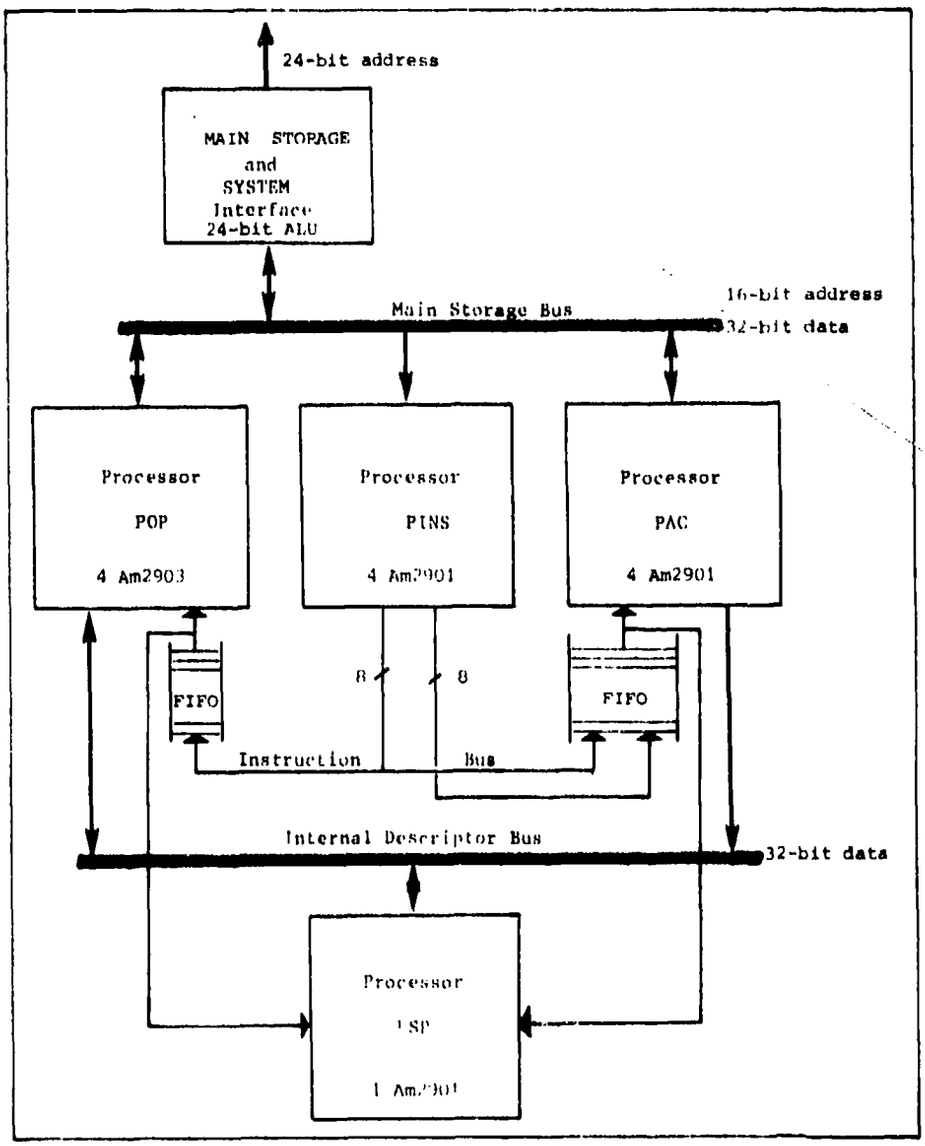


Figure 5: PASC-HLL internal architecture

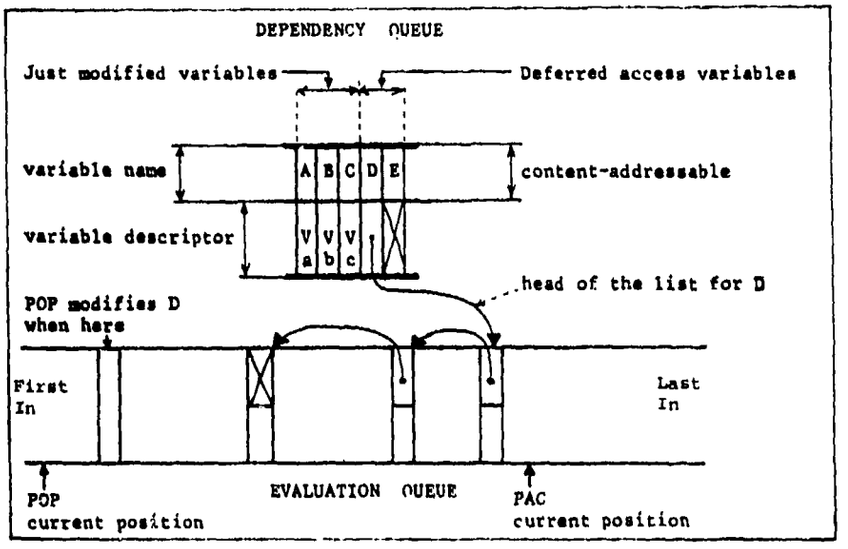


Figure 6

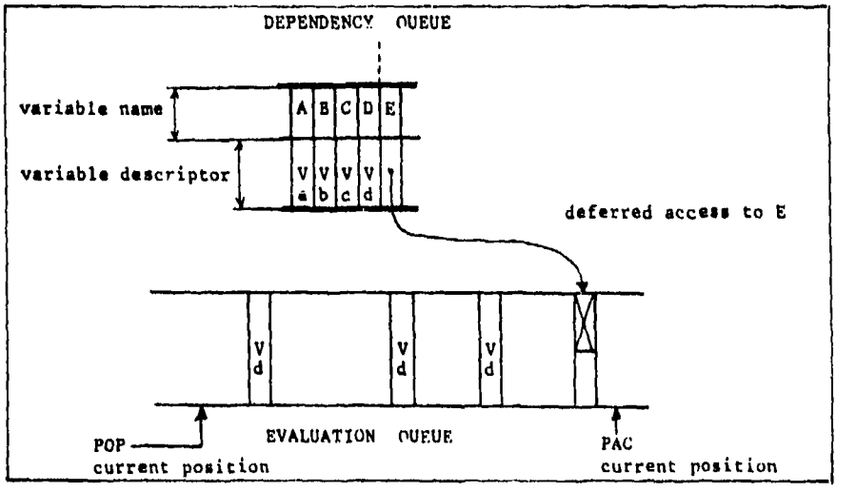


Figure 7

AN EXTENSIBLE STACK-ORIENTED ARCHITECTURE FOR A HIGH-LEVEL LANGUAGE MACHINE

Robert P. Cook* and Insup Lee

Computer Sciences Department and
Mathematics Research Center
University of Wisconsin-Madison
Madison, Wisconsin 53706

Abstract

MCODE is a high-level language, stack machine designed to support strongly-typed, Pascal-based languages with a variety of data types. The instruction set is constructed for efficiency and extensibility and is based on an examination of common programming language operations. The architecture provides programmed control over both operand type selection and address field widths. In addition, right operand addressing is included to improve the size characteristics of MCODE instructions over those of traditional stack machines. The design is compared for efficiency with the instruction sets of the EM-1, Digital Equipment PDP-11 and VAX-11/780.

CR Categories: 4.12, 4.22, 4.9, 6.21

Keywords and Phrases: stack machine, computer architecture, addressing modes.

1. Introduction

With the growing use of high-level languages for systems and applications programming, computer instruction set design has moved from bit selection of internal CPU data paths to instruction sets which are oriented to common high-level language operations. Tanenbaum[10] discusses a stack machine(EM-1) designed with this philosophy. The EM-1 is intended to directly execute the code produced by the SAL compiler. SAL is a typeless systems programming language similar to BCPL[9]. In this paper, we have extended the EM-1 to provide an instruction set for a Pascal-based, strongly-typed, systems programming language, Modula[12], which was designed by Wirth and implemented by Cook[6]. Our Modula machine code, MCODE, not only provides extensible type operations but also maintains the efficiency of the EM-1. The EM-1 was designed based on an analysis of 300 procedures comprising 10,000 lines of code. The MCODE improvements are based on our analysis of 5,000 Pascal procedures with over 160,000 lines of program text.

The next section gives a brief EM-1 description which is followed by a discussion of the MCODE improvements. Also, the instructions used for expressions and Modula statements are illustrated. Finally, some comparisons are drawn with respect to other current architectures, including the the PDP-11[1] and VAX[2].

*The author is partially supported by U. S. Army contract DAAG29-75-C-0024 and National Science Foundation grant MCS-7903947.

2. Background

Tanenbaum designed the EM-1[10] to optimize the most frequently occurring high-level operations in programs as analyzed by himself, Knuth[8], Alexander and Wortman[3], and Wortman[13]. The most effective innovations in the EM-1 are encoding references to the first 12 bytes of local procedure storage and 8 bytes of static storage as single opcodes, array element accessing, and "if" statement comparison and branching. The hypothesis is that smaller code sizes will enhance faster program execution by better utilizing the bandwidth of CPU data paths. In addition, as the machine gets closer to the source language, compilers can produce more efficient code and can eliminate space-consuming peephole optimization routines.

Another important aspect of the EM-1 design is the notion of giving the programmer code improvement tools which are machine independent. In Knuth's Fortran analysis[8], he strongly suggested that program execution histories be automatically generated for each job. With Tanenbaum's machine organization, the programmer need only declare the most frequently used variables first in textual order to effect a performance improvement.

3. Extensions

The first problem that we found in trying to use the EM-1 design was its lack of a variety of data types. Modula provides the user with character, Boolean, long and short integer, and floating point operations. When the EM-1 is extended to encompass these operations, the 255 opcode limit is quickly exceeded. Our solution was to introduce modes of computation. A mode sets the CPU's fetch and execute mi-

croprogram to adapt to a particular data type such as floating or integer. A collection of 8-bit opcodes is provided to set the CPU mode. Therefore, a single "+" opcode suffices for all addition operations on any data type. The setting of the mode can be thought of as the replacement of the microcode jump table for a subset of the opcodes.

The mode approach is based on our observation that expressions are usually comprised of operands of the same type; thus, we expect that the space occupied by any extra instructions needed to set the mode will be offset by the savings in opcode space. Modes also provide an expansion and contraction capability for machine families. For instance, all floating point operations could be eliminated to build microprocessors intended for traffic control or a decimal mode could be added for commercial applications. For many environments, the savings in microcode space could be significant.

Our second extension was to provide direct addressing for right operands. According to all of the analyses, expressions tend to be simple. Tanenbaum found, for instance, that 31% of all assignment statements had a single term for a right hand side. Consider the evaluation of "a+b" on a typical stack machine. We must "push a", "push b", "pop b and add", and "replace a with result". The alternative is to "push a", "add b", and "replace a with result". This sequence not only saves an instruction fetch but also the redundant push and pop of "b" plus the instruction space. These savings will be replicated for every term in any expression which can be evaluated from left to right.

Finally, we have extended Tanenbaum's single byte addressing modes, provided an option to shorten address fields, improved subscripting, record and pointer referencing, and introduced some additional high-level language oriented constructs. In the next section, we will discuss operand addressing.

4. Operand Addressing

The three MCODE instruction formats are illustrated below:

```
        FORMAT 1:
FORM 2,3,3      0,opcode,local address

        FORMAT 2:
FORM 8          opcode [operands]

        FORMAT 3:
FORM 8,8        255,opcode [operands]
```

In MCODE, addressing is partitioned into references to either static or local procedure storage. The MCODE machine uses byte addressing and has an address space of 2**32 bytes. The instruction formats are designed so that the most frequently occurring operations require a minimum of instruction space.

A format 1 instruction can address the first 8 16-bit words of the current procedure's activation record. The impact of this convention can be seen by noting that our results indicate that 97% of all procedures have fewer than 4 formal parameters and 90% of all procedures have fewer than 4 local variables. Tanenbaum's short address convention for static variables was eliminated since the size of the static address space is not known until load time. However, the number of parameters and local variables is known at compile time. In addition, our analysis shows that 54% of all variable references were to local variables or parameters. To test the effect of this idea, we changed all the local variables in the Modula compiler to C[7] "register" variables which decreased each instruction reference by 16 bits. The compiler's size decreased by 10% and its compile rate went up several hundred lines per minute.

The format 2 and 3 instructions can have their operands on the stack or can have a right operand specification. Operand addressing is optimized in a fashion similar to that provided by the B1700[11]. The AMODE instruction sets the address field width to 8, 16, or 32 bits for references to either static or local storage. Note that program counter relative addressing is not affected. More than 90% of all Modula programs can use an AMODE which selects 8-bit local and 16-bit static addresses.

As an example, the 8-bit AMODE setting would save 8 bits per operand reference over the 16-bit addresses used in the PDP-11. The AMODE setting has no effect on indirect addressing on the stack. The VAX implements 8-bit address fields but an 8-bit selector is also required for a total of 16 bits.

A natural concern, however, is keeping AMODE set correctly. Since Modula has no "go to", the AMODE bookkeeping is easily maintained on the parse stack. Also, the procedure call instructions automatically save and restore mode information. In addition, the linkage editor is responsible for checking address field overflow if too small an AMODE is being used. MCODE implements the following addressing forms:

```
A       operands on the stack
B       {static|local}x{direct|indirect}
C       local direct
D       indirect address on the stack
E       32-bit absolute address
F       constant(8, 16, 32 bits)
G       constant(0-15)
H       {subscript|element} x B
        subscript-((sp↑)-1)*Mode size + EA)
        element  -((sp↑)+EA) Effective Addr.
```

```
I       local x {direct,indirect,indirect x
                 {subscript,element}}
J       8-bit jump offset
K       16-bit jump offset
```

Forms B and H cover accesses to simple variables, pointers, one dimensional arrays, and record elements occurring in static and local storage, or as parameters. Subscript addressing assumes a lower bound of one which is the most common case. For direct addressing, different lower bounds can be subtracted from the address field to produce the correct subscript calculation. Forms F and G are used for immediate addressing while forms E, J and K are used for program counter relative jumps and absolute addressing. Forms I and C are used with the format 1(8 bit) instructions. Form I can be used to access local variables, "const" simple parameters, "var" simple parameters, and array and record parameters.

Tanenbaum[10] recommends that references to global procedure variables be implemented by a microcode search of the procedure call back-chains. The claim is that this method eliminates the overhead of maintaining a static display. Based on our experience with implementations of Algol[5] and Pascal[4], a single reference to a global variable uses more time than that needed to update the display. The following code sequence is typical.

```
procedure entry:
        CONTROLBLOCK[SAVE]=DISPLAY[NEST]
        DISPLAY[NEST]=PB

procedure exit:
        DISPLAY[NEST]=CONTROLBLOCK[SAVE]
```

The first ten locations in static storage are used for the DISPLAY. According to our study, 85% of all procedures were not nested; 11% were nested one level; and 4% were nested 2 or more levels. Out of the 5,000 procedures that we examined, one was nested to 4 levels. Therefore, a maximum of ten nesting levels was considered sufficient. Next, we will examine the format of the one byte instructions.

5. Local Variable References

We followed Tanenbaum's design by allocating 64 opcodes to special addressing. As we discussed previously, the local variable address space was set at 8 16-bit words, or a 3-bit address field. This left 3 bits for opcodes. These 8 opcodes were partitioned as follows:

```
PUSH    Form I          (sp↓) = (EA)
POP     Form C          (EA) = (sp↑)
ADD     Form C          (sp) += (EA)
SUB     Form C          (sp) -= (EA)
CMPB=   Form C,K        if (sp↑)=(EA) then
                          (pc) += SE(K)
CMPB<>  Form C,K        if (sp↑)<>(EA) then
                          (pc) += SE(K)
```

The PUSH instruction uses two opcodes for direct or indirect references to simple variables, and two opcodes for indirect, or "var", references to arrays and records. The number of addressing modes for POP was decreased to one in order to increase the number of opcodes. In addition, we found that variable loads occur in a 2.7/1 ratio over variable assignments which indicates that POP is used less frequently than PUSH. The last four opcodes were assigned based on our frequency of use information. Out of all operator occurrences, "+" was used 21% of the time, "-" was used 9%, "=" was used 20%, and "<>" was used 10% of the time. According to Tanenbaum, the dynamic frequency of these operators is even higher. In conditional expressions, we found that "=" made up 33% of all operators and that "<>" was used 17% of the time. Since Tanenbaum found that "if", "repeat", and "while" had a dynamic frequency of 38%, the comparisons were implemented to both test and jump. Using these formats, many subprograms can be completely coded using only 8 bit instructions.

6. Right Operand Addressing

Because of the number of opcodes needed for right-operand addressing, we restricted the operators based on the same frequency analysis which was used to select the 8-bit instruction set. The following table lists the instructions which can address memory:

```
PUSH    Form A,B,D,F,H,G    (sp↓) = (EA)
POP     Form A,B,D,H        (EA) = (sp↑)
PUSHA   Form B,E,H          (sp↓) = EA
ADD     Form A,B,F          (sp) = (sp)+(EA)
ADDTO   Form B              (EA) = (EA)+(sp↑)
AND     Form A,B,F          (sp) = (sp)&(EA)
CLR     Form B              (EA) = 0
CMPB=   Form A,B,F          if (sp↑)=(EA)
CMPB<>  Form A,B,F          if (sp↑)<>(EA)
DEC     Form B              (EA) = (EA)-1
INC     Form B              (EA) = (EA)+1
MUL     Form A,B,F          (sp) = (sp)*(EA)
SUB     Form A,B,F          (sp) = (sp)-(EA)
SUBFM   Form B              (EA) = (EA)-(sp↑)
```

The selected operators make up 80% of all operator references in the Pascal programs that we analyzed. Address modes B and F were chosen since 35% of all operand references were to simple variables and 36% of all operands were constants. The ADDTO and SUBFM instructions correspond to Modula statements.

7. Array, Record and Pointer References

Simple record references are treated

just like simple variable references and can be accessed using direct addressing. However, arrays of records or records as parameters must be accessed by an offset from a base address. The "element" address mode implements the pointer or parameter case.

Our analysis showed that 20% of all array references had a single constant subscript and that 60% of all subscripts were a single variable. The constant subscript case resolves to a variable address so the standard address formats can be used to access the array. The "subscript" mode was introduced to implement accesses to one dimensional arrays. In fact, we found that references to multidimensional arrays made up only 10% of all array references. MCODE uses descriptors to implement the multidimensional case.

In the EM-1, every array has an array descriptor cell, an array descriptor packet and an array data area. This approach works fine for Algol but not for Pascal-like languages. First in Pascal, all arrays have static bounds so a single descriptor can be generated in static storage. This approach allows descriptors to be shared and saves stack space as well as setup time. Secondly, Pascal allows arrays of arrays and pointers to arrays which implies that the base address of an array may already be on the stack and not in a descriptor. The MCODE SUBS instruction transforms the subscripts into a single byte offset which can then be used by the PUSH or POP instructions. The SUBS instruction also checks each subscript for validity.

```
SUBS            descriptor address
```

The instruction address points to an array descriptor which contains the number of bounds, bounds pairs, multipliers, element size and virtual origin. SUBS leaves the element index on the stack. For instance, "A[I].B[J]" would produce the following code.

```
PUSH    I
SUBS    A desc.
PUSHA   element( A+B offset)
PUSH    J
SUBS    B desc.
ADD
```

For most Modula programs, each array type can be described by a single instance of a descriptor no matter how many variables of that tyoe are created. Next, the expression operators will be described.

8. Operators

The following table lists the MCODE operators which are all format 2 instructions.

```
ABSolute            LoGical Shift
ARith. Shift        MOD
CONVert             NEGate
DECrement           NOT
DIVide              OR
DUPlicate           SQuaRe
INCrement           XOR
```

MCODE also includes instructions for moving and comparing blocks of storage as well as library call instructions to implement the Modula virtual machine environment and the floating-point math routines. In the next section, the code generated for the "case", "if" and "for" statements will be discussed.

9. Statements

Procedure call and return are very similar to the EM-1, except for the display updating, and will not be described. The "if" statement is implemented with the following instructions:

```
CoMPare             = > < >= <= <>
CoMPare Branch      = > < >= <= <>
Branch              =0  <>0
Branch
```

As an example, the statement "if a<>b then inc(a) end" would generate the following code:

Instructions			Size	PDP-11		Size
PUSH	a		8	CMP	a,b	48
CMPB=	b	L1	24	JEQ	L1	16
INC	a		16	INC	a	32
			48			96

The syntax and code generated for the "for" statement are listed below.

```
for v:=e1 by e2 to e3 do S end
        PUSHA   v
        PUSH    e1
        PUSH    e2
        PUSH    e3
        FOR     L2
L1      S
        ENDFOR  L1
L2
```

The "case" instructions are as follows:

```
CASE            constant, offset
CASE            constant, constant, offset
CASETBL         constant, constant
```

These three forms cover the situations in which the "case" is distinguished by a single value, a range of values, or a jump table. Next, we will analyze the effectiveness of MCODE with respect to other machine designs.

10. Comparison with Other Machines

The results in Figure 1 extend the table in Tanenbaum[10] to include the VAX and MCODE. Obviously, the special addressing and descriptor-based array computations make a significant difference. MCODE performs better than the EM-1 for expressions and parameter referencing and is as good in all other areas. The difference in the "if" tests occurs because the EM-1 assumes a 2-bit field for branch offsets while we used an 8 bit field. The VAX instructions are computed using 8 bit displacement addressing. In addition, it should be pointed out that the VAX and MCODE are supporting many more data types than the PDP-11 or the EM-1. Figure 2 recomputes the space for the same statements but with all the machines forced to use 16 bit addressing.

The values in Figure 2 give a lower bound on the performance of MCODE whereas Figure 1 gives an upper bound on the difference. For 16-bit addressing, which would be used for references to static storage, MCODE is better in all categories. The EM-1 is forced to use a 16-bit opcode to access 16-bit addresses which results in its poor performance. Since 47% of all variable references are to static storage, we feel that this improvement could have a significant impact on execution speed. The VAX is still quite poor with respect to subscripting even though a special instruction is available for that purpose. Also, the figures do not reflect the dynamic effect of the savings since Tanenbaum's measurements indicate that the Figure 1 results are even more significant at runtime.

11. Conclusions

We feel that the availability of modes as an extension mechanism for high-level language machines can be a significant factor in adapting microprocessors to changing environments. Also, modes contribute to space efficiency in the instruction set. The use of address mode settings to reduce address field sizes and right operand addressing also contribute space savings. The current version of Modula produces PDP-11 or VAX code so we have the means to compare the exact statistics on the static and dynamic behavior of MCODE with these machines using the same programs in the same environment. Our analysis should contribute to the alternatives available for opcode design in modern machine families.

REFERENCES

[1] PDP-11 Processor Handbook. Digital Equipment Corporation, (1975).

[2] VAX11/780 Architecture Handbook. Digital Equipment Corporation, (1977).

[3] Alexander, W.G., and Wortman, D.B. Static and dynamic characteristics of XPL programs. Computer 8, (1975), 41-46.

[4] Burger, T.M. A portable optimizing Pascal compiler. M.S. Thesis, Vanderbilt University, (1978).

[5] Cook, R.P., Hansen, G. and Haynam, G. Extended ALGOL 60 reference manual. Vanderbilt University Computer Center Report, (1970).

[6] Cook, R.P. An introduction to modular programming for Pascal users. University of Wisconsin Technical Report 372, (Nov. 1979).

[7] Kernighan, B.W. and Ritchie, D.M. The C Programming Language. Prentice-Hall Inc., (1978).

[8] Knuth, D.E. An empirical study of FORTRAN programs. Software--Practice and Experience 1, 1(1971), 105-133.

[9] Richards, M. BCPL: A tool for compiler writing and systems programming. AFIPS SJCC V. 34, AFIPS Press, Montvale, N.J., (1969), 557-566.

[10] Tanenbaum, A.S. Implications of structured programming for machine architecture. Comm. ACM 21, 3(March 1978), 237-246.

[11] Wilner, W.T. Design of the Burroughs 1700. AFIPS FJCC V. 41, AFIPS Press, Montvale, N.J., (1972), 489-497.

[12] Wirth, N. Modula: A language for modular multiprogramming. Software--Practice and Experience 7, 1(Jan. 1977), 3-35.

[13] Wortman, D.B. A study of language directed computer design. Technical Report CSRG-20, U. of Toronto, (1372).

Figure 1

Direct Addressing Instruction Size(in bits)

Statements	MCODE	EM-1	PDP-11	VAX
i:=0	16	8	32	24
i:=3	16	24	48	32
i:=j	16	16	48	40
i:=i+1	16	8	32	24
i:=i+j	24	32	48	40
i:=j+k	24	32	96	56
i:=j+1	24	24	80	48
i:=a[j]	24	32	128	104
a[i]:=0	32	32	112	88
a[i]:=b[j]	40	48	192	168
a[i]:=b[j]+c[k]	64	80	304	248
a[i,j,k]:=0	64	48	176	200
if i=j then	32	24	64	64
if i=0 then	24	16	48	48
if i=j+k then	40	40	112	96
if flag then	24	16	48	48
call p	24	16	64	32
call p(i) value	32	24	96	56
call p(i,j)	40	32	128	80
call p(i) byref	40	32	112	56
for i:=1 by 1 to N do a[i]:=0 end	104	88	176	116

Figure 2

16-Bit Address Fields

Statements	MCODE	EM-1	PDP-11	VAX
i:=0	24	32	32	32
i:=3	32	48	48	40
i:=j	48	64	48	56
i:=i+1	24	32	32	32
i:=i+j	48	104	48	56
i:=j+k	72	104	96	80
i:=j+1	56	80	80	64
i:=a[j]	72	96	128	128
a[i]:=0	64	72	112	104
a[i]:=b[j]	96	128	192	200
a[i]:=b[j]+c[k]	152	200	304	296
a[i,j,k]:=0	128	136	176	232
if i=j then	64	96	96	80
if i=0 then	48	64	80	56
if i=j+k then	88	136	160	120
if flag then	48	64	80	56

The High Level Language Instruction Set of the SYMBOL Computer System

Robert F. Cmelik†

David R. Ditzel†

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The instruction set for the SYMBOL computer system is discussed in detail. The SYMBOL computer is a large scale multiprocessor which implements a high level language, compiler, text editor and time-shared operating system entirely in hardware. The intent of the paper is to document the instruction set, as used in the working system for over seven years. Covered are the internal codes, what they do, and the associated machine maintained data structures

Introduction

The SYMBOL computer system[1,2,3] is of great importance in the field of computer architecture since it represents a major departure from von Neumann architectures and is one of the few examples of an experimental (or commercial) machine that resulted in a full scale working High Level Language Computer System. Although the high level SYMBOL Programming Language (SPL) was implemented in the machine without the aid of software, SYMBOL does have an internal instruction set much like any computer. Unlike most computers, however, the SYMBOL Instruction Set is non-von Neumann and at a very high level, with almost a one-to-one mapping between tokens in the source code and instructions in the object code. Though the instruction set is probably the best way to describe the computational abilities of SYMBOL, it has been one of the least documented features. This paper seeks to fill this gap by providing a detailed description of the instructions, how they are executed, and the internal data structures used in executing SYMBOL object programs.

SYMBOL Architecture Overview

Because of SYMBOL's unusual machine architecture, a brief description of the system is in order. The SYMBOL computer system is composed of eight relatively autonomous processors: the System Supervisor, the Input/Output Processor, the Channel Controller, the Drum Controller, the Memory Reclaimer, the Memory Controller, the Translator, and the Central Processor. The last three of these are of special interest for the purposes of this paper.

Program execution is controlled by the Central Processor, which is itself composed of four sub-processors. The Instruction Sequencer is responsible for fetching instructions, executing some directly and delegating the rest to another sub-processor. The Arithmetic Processor[4] performs traditional arithmetic operations with precision controlled, decimal arithmetic. The Format Processor[5] handles character oriented operations, as well as the packing and unpacking of numbers. Lastly, the Reference Processor controls all identifier referencing.

One of the more unusual aspects of SYMBOL is that the memory structure is not organized as a contiguous set of sequentially numbered storage cells. Instead, storage is viewed by most of the processors as a limitless supply of variable length storage strings, whose storage cells (machine words) are logically sequential but may not be consecutively addressed in memory. The SYMBOL memory structure consists of four hierarchal levels. At the lowest level a core memory and a rotating magnetic drum constitute the physical memory. Next is a paged virtual storage system consisting of 2^{24} 64-bit words, of which 4096 2K-byte pages were implemented. The Memory Controller, with a set of high level memory operations, maps virtual storage into "logical storage."[6,7] At the highest machine level are the operations which operate on the user data structures. Throughout the rest of this paper the word "string" referring to storage means a separate and logically sequential series of words, used in the same manner as segments in other computer systems.

The SYMBOL Programming Language

Because the instruction set of the SYMBOL machine is so directly tied to the language it implements, the reader will find the following sections easier to understand by referring to one of the many descriptions of the language.[3,8,9,10,11] Basically, SPL is a general-purpose procedural block-structured language. In many ways, it can be viewed as a mixture of APL, ALGOL and LISP. The language is free of most declarations as to the size or type of data objects; these attributes can vary dynamically during the life of a program. Data objects are either scalars (i.e. a sequence of characters that may happen to fit the definition of a number, Boolean, or string), or the object is a structure whose elements are either scalars or other structures. Structures may be of any arbitrary shape which is representable by a tree, and may not be recursively defined. Procedures pass parameters via call-by-name, also known as call by substitution. There are no automatic variables; all variables are statically allocated. Scoping rules are such that a variable is known only locally, unless explicitly declared to be global. SPL also has ON blocks, similar in may ways to ON blocks in PL/I.

Instruction Set Overview

SYMBOL instructions are ordered in reverse Polish notation and make use of an expression evaluation stack. SYMBOL uses both descriptors and tags for recording the attributes and structure of data. Descriptors are grouped together in Name Tables, generated by the Translator at compile time. Type tags are associated with the data, at the beginning of a data object the tag records the type; a tag is also used to denote the end of a data object. The basic instruction set is shown in Table 1, with the internal bit representation shown in hexadecimal. Throughout this paper internal codings will be shown in hexadecimal unless otherwise indicated. Addresses in SYMBOL are 24 bits long and address sixty-four bit words. For hardware bussing simplicity, each word contains a maximum of two instructions, each half-word instruction consisting of an eight bit opcode followed by a twenty-four bit address field. Only six of the opcodes require an address.

Internal Representation of Data Values

The storage format for scalar character string values consists of a String Start character (F5), followed by the characters in the string in ASCII, followed by a String End character (F6); this is called the data string format. Scalar values appear in the object string as a result of literals in the source string. A scalar value may be stored in a Name Table if it is six or fewer characters in length (since there must be room for the string, start and end characters in an eight character word). For longer strings, a memory string is allocated, and a pointer to this string is placed in the Name Table. However, if the string should later shrink to six or fewer characters, it would remain in the memory string, and not be placed in the Name Table.

† Work done at Iowa State University under NSF grant GJ34097X

Table 1. SYMBOL Instruction Set

	8	9	A	B	C	D	E	F
0	Input	Block†	To	From	Reclaim Group	Name† Table Pointer	Link to Simple Variable	Numeric + +
1	Output		Data		Assign Group		Link to Structure	Numeric + −
2	Disable				Insert Group			Numeric − +
3	Enable		String		Fetch Terminal Header		Link to Label	Numeric − −
4			Exact		Fetch and Follow	Direct† Parameter	Link to Data in Name Table	Numeric True Zero
5		Go To	Empirical	If False† Then Jump	Fetch Reverse	Indirect† Parameter	Link to Unresolved Subscript	String Start
6		Pause			Follow and Fetch		Link to Temporary Data	String End
7	Return	System		End Block	Fetch Direct	Transfer†		
8	Before	Limited		In	Store and Assign	Parameter Return	Link to Structure Field	
9	Same	Abs			Store Only	Source Pointer		
A	After	Lte	*	:	Store and Insert	Integerize	Link to Field (In)	
B	Not	Gte	+	End Statement	Store Direct	Negate	Link to Structure(In)	
C	And	Less	,	Limit	Delete to End	Field Mark	Link to Substructure	Begin Vector
D	Or	Neq	−	Equal	Delete String	Perform Subscription		Begin Structure
E	Join	Greater		Format	Delete Page List		Link to Data In N.T.(In)	End Vector
F	Mask	→		Interrupt	Store Terminal Header	←		End Structure

† Requires address field.

A second storage format is used for packed decimal numbers, the numeric field format. This is the format in which the Arithmetic Processor produces its results. If an operand for an arithmetic operation is in data string format, the Format Processor will automatically convert the operand into the numeric field format before the Arithmetic Processor proceeds with its operation. The components of a number stored in numeric field format are the exponent sign, the mantissa sign, the exponent magnitude, the mantissa, and the precision code. The exponent and mantissa signs appear as two bits of the start character (F0, F1, F2, or F3). The character following the start character contains the exponent magnitude as two BCD digits. The 1 to 99 digit mantissa is stored after the exponent character and occupies as many words as are required at ten digits per word. (The first two and last bytes of a word are not used for packed BCD data so that mantissa digits will always occupy the same portion of the word.) Following the last mantissa digit is a four bit precision code which indicates that the number represented has either infinite precision (1111), or only that precision implied by the number of mantissa digits (1110). The representation for a true numeric zero starts with an F4. The last word of the numeric string is indicated by a set, high order bit in the last byte.

Structure values may appear on the stack or in the object string in linear format, or elsewhere in tree format.[12] In tree format, a structure is stored in a memory string as a succession of scalar values and links to substructures. The scalars are stored with start and end characters as described above, and are aligned on word boundaries. If, at a later time, the scalar expands and requires more space, then additional (64 byte) memory groups are linked (inserted) into the memory string. A link to a substructure consists of a single word beginning with the character EC, and containing an address pointing to a separate memory string where the substructure begins. Following the last component in the structure, and in each substructure, is the End Vector character (F7). A null vector, the analogue of the null string, is stored simply as a memory string beginning with an F7.

In the linear format, the structure value begins with a Begin Structure character (FD), and ends with an End Structure character (FF). Between these two characters are stored the components of the first level of the structure. Scalar components are stored with start and end characters, aligned on word boundaries as in the tree format. If the component is a substructure, however, it is represented by a a Start Vector character (FC), followed by the components of the substructure (which may be scalars or structures), followed by an End Vector

character (FE). The start and end characters FC, FD, FE, and FF of the linear format are the only essential characters in the words they begin, so seven bytes are always wasted in these words.

If an initial value statement in SPL is preceded by the keyword SWITCH, the Translator treats the initial values that are being assigned as identifiers for labels. The Translator stores values in the object string as it would for an ordinary initial value statement, using a single word to store each scalar value. The scalar label value is stored as a word beginning with the character D0 (which is also the opcode for the Name Table Pointer instruction), followed by a 24-bit address pointing to a data descriptor for the label in a Name Table, followed with the character F6. Label values may be "moved around" like other values (e.g. assigned to variables, passed as procedure arguments, returned as function values, etc.). And of course, a label value may be the operand of a Go To instruction. Label values cannot appear as operands for any arithmetic, string or Boolean operations.

The Evaluation Stack

For each active instance of a block, the Central Processor maintains a stack to be used for evaluating expressions, procedure calling, and passing information to other processors. (Whenever possible, the Central Processor keeps the top word of the stack in an internal register.) Each stack is a unique memory string and is created when a block is entered, and deleted when the block is exited.

The first three words of the stack are a save area for the block. When the stack is created, a pointer to the block's Name Table, and a pointer to the calling block's stack are stored in the first word of the stack. If this block should call another block (explicitly by a procedure call or implicitly by an ON reference), then a pointer to the start of the called block's object string and the contents of the status register are stored in the second word. Also, the return point and top of stack pointer are stored in the third word of the stack.

The remainder of the stack is for expression evaluation. If an operand is being pushed on the stack and it is one word or less in length, it is copied directly to the stack. Otherwise, it is left in place, and a pointer (link) word is stacked. (The two exceptions to this rule are operands for the Output operation, and values for the assignment operations.)

Link words begin with a character indicating the nature of the operand: E0 (simple variable or value), E1 (structure), E3 (label), E4 (scalar value in Name Table), E5 (memory string containing subscripted variable reference), E6 (scalar value that won't fit in one word), E8 (scalar or structure valued component of a structure), EA (IN reference to simple variable), EB (IN reference to structure), or EE (IN reference to variable with value in Name Table). The code generated for an IN expression is the IN instruction (B8), followed by a subscripted variable reference. The result of the expression is a Boolean value indicating whether the indicated component of the variable exists. The left address field of the link word is used when pointing at the data value and the right address field is used when pointing to the descriptor for the value.

The Colon (BA), Integerize (DA), and Perform Subscription (DD) instructions are related to structure references. Expressions for subscripts are evaluated (using the stack for intermediate values) and are then converted to a four digit integer. Each subscript is stored on the stack in a word beginning with BA or DA, which indicates the type of subscript. DD (Perform Subscription operation) follows the last subscript. The character substring operation is handled as a form of subscription. (For example, in SPL x[1,2:3] is the 3 character string from the first component of x, beginning at character position 2.) The subscript preceding the colon is stacked in a word beginning with the character BA (Colon operator). All the remaining subscripts are stacked in words beginning with DA (Integerize operation). After the Perform Subscription operator has been stacked, the link to the variable being subscripted and the subscript list are moved to a new memory string and a link word (E5) is placed on the stack pointing to the string. The subscripted reference is not evaluated until it is absolutely necessary to have the value or location indicated.

Name Tables

The Translator produces a Name Table for each block (main program, procedure or ON block) in the source string. All references made in that block to labels, procedures, or variables are made through the descriptors in that block's Name Table. Figures 1 shows the organization of the Name Table and figure 2 shows the organization of the control words. The first word of a Name Table is called the Block Control Word. It contains two address fields which are used to link all the Name Tables for a program. The first address field is used to forward link all the Name Tables together in a single list, beginning with the Name Table for the main program. The second address field is used as a pointer to the Name Table for the statically enclosing block. (The Block Control Word for the Name Table of the main program which is the outermost block, has no such pointer.) The actual bit definitions are shown in Table 2.

The Block Control Word contains a bit indicating block in use, and a bit indicating block recursed. When a block is entered, the block-in-use bit is set, and the bit is cleared when the block is left. If a block is reentered (i.e. entered when the block-in-use bit is already set), the Central Processor calls on a software routine to perform a "fixup" to handle recursion.[13] (The hardware was not specifically designed to handle recursion.) This software must create a copy of the block's Name Table, modify the original Name Table to initialize local variables, and then set the block-recursed bit. Similarly, if a block is left with the block-recursed bit set, another software routine is called that must undo the work of the first routine.

Following the Block Control Word is a succession of entries for each identifier in the block. Each entry consists of the ASCII name of the identifier, taking as many eight-byte words as necessary and padding with nulls, and followed by a one word data descriptor for the identifier called the Identifier Control Word (IDCW). Table 3 shows the bit layout of the IDCW.

If the identifier is a local variable, it is so tagged in the IDCW. There are also flag bits to indicate whether the variable is scalar valued, structure valued, or has not yet been assigned a value. If the value is undefined, then the first time that the variable is accessed, it is given a null, scalar value, which is interpreted as a zero for arithmetic operations. If a value is defined, then that value may appear in the object string, the Name Table (scalars only), or elsewhere in memory.

If an initial value statement occurs in the source string, the Translator will place a pointer to the object string location of the value in the IDCW for the variable. When the variable is first accessed, the value is copied from the object string. (Note that for structures this means converting from linear to tree format.) A pointer to the value replaces the old pointer in the IDCW, and the bit indicating data in object string is cleared.

For data in Name Tables, recall that scalars (excluding label values) are stored with a start character which begins with hex "F". This half byte at the beginning of an IDCW is used to indicate data in Name Table (i.e. the IDCW is the scalar value itself). If the value is elsewhere in memory, the IDCW contains a pointer to the memory string where the value begins.

Global variables are variables that are known in outer blocks. SYMBOL permits the nesting of blocks as does PL/1. In contrast to PL/1 however, variables are local unless declared global in an SPL Global statement. The IDCW contains a bit indicating if the variable is global, and a pointer to the IDCW for the variable in the enclosing block. There may in general be many levels of such indirection. If the identifier is for a procedure, the global variable and procedure bits of the IDCW will be set. Procedures are an exception to the local unless declared global principle. A procedure's scope always extends to inner blocks. The IDCW will also contain a pointer to the location in object string where the procedure begins. In SPL, a label is always

Figure 1. Name Table Format

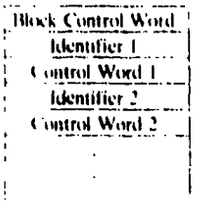


Figure 2. Name Table Control Word Fields

flags	left address field	flags	right address field
0 78	3132	3940	63

Table 2. Block Control Word Format

Bit Position	Meaning
0	Control word (always 1 for BCW)
1	Start of Name Table (always 1 for BCW)
2	End of Name Table
4	This block translated in privileged mode
5	Global linking done
6	Left address field valid
7	Right address field valid
8-31	Forward link in list connecting all Name Tables
37	Block in use
38	Block recursed
40-63	Backward link to BCW of enclosing block
Others	Unused

These bits are used by Translator and not by Central Processor

Table 3. Identifier Control Word Format

Bits	Meaning
0	Control word (always 1 for IDCW)
1	Start of Name Table (always 0 for IDCW)
2	End of Name Table
3	New. Set for local variables and labels, and for procedures
0-3	Indicates variable with value in Name Table when all four bits are set. The IDCW is a scalar value (not allowed for last IDCW in Name Table)
4	Global. Set for global variables, labels, and procedures, and for direct formal parameters
5	Variable with value in object string
6	Variable is structure valued (if 1), scalar valued (if 0)
7	Flag bits 32-39 valid
8-31	For variables, pointer to value (zero if space not yet assigned). For label or procedure, pointer to object string entry point
32-39	If bit 7 is 0, two digit current pointer index used during subscription If bit 7 is 1, then bits 32-36 are:
32	ON block feature enabled
34	Identifier is a label
35	Identifier is a procedure
36	Identifier is a indirect formal parameter
40-63	For identifiers with ON blocks, pointer to ON block code. For structures without ON blocks, current pointer address
Others	Unused

local to the block where it occurs. However, the scope of the label may be extended to an inner block if that block contains a Global statement naming the label. Thus a Go To can be used to jump out of a block. The IDCW for a label contains a pointer to the location in the object string where execution is to continue.

For procedures, the first entries in the Name Table will be the formal parameters (if any), simply because they are the first identifiers encountered by the Translator when scanning the source string. SYMBOL implements call-by-name for all parameters. When a procedure is called, the formal parameters are linked. Two mechanisms exist in SYMBOL for linking parameters. The most general method (indirect parameter) is to compile code near the calling point in the object string to evaluate the actual parameter (commonly known as a thunk). When the procedure is called, a pointer to the code for the actual parameter is placed into the IDCW for the formal parameter. Whenever the formal parameter is referenced, this code is executed and the actual parameter is left on the top of the stack. (Part of the fixup for recursion requires that a modified copy of this code be generated, since it will in general, contain absolute address references to the original Name Table.) Often, however, the actual parameter is a simple variable.[14] In the second mechanism (direct parameter), when the procedure is called, the IDCW of the formal parameter is set up as if it were a global variable with a pointer to the IDCW of the actual parameter. The Translator determines which mechanism to use and produces the appropriate instructions. Often the Translator chose to compile an indirect parameter where a direct parameter would suffice because it was too stupid.

SYMBOL provides a mechanism for trapping references to variables, procedures and labels, called the ON block. The IDCW for an identifier which has an ON block contains a pointer to the object string for that ON block and a bit indicating whether or not the option is in effect. This bit, which is initially set, may be cleared by a Disable instruction (82) or set by an Enable instruction (83). If the identifier is a variable name, the ON block will be invoked immediately after an assignment to that variable occurs. If the identifier is a label, the ON block will be invoked upon encountering a Go To statement to that label before the transfer actually takes place. If the identifier is a procedure, the ON block will be invoked upon encountering a call to that procedure before entry takes place.

There is one more piece of information stored in the IDCW. Recall that a vector stored in memory is a succession of arbitrarily long scalar values placed end-to-end. Because the addresses of a component of a vector can not be calculated, finding the n'th component would mean scanning the preceding n-1 components. One of the mechanisms used to speed up this search is called current pointer. In the IDCW for the structure is stored the subscript used in the last reference, and the address of that component in memory. If the next reference is to a component succeeding the last, the search begins where the last search left off. The mechanism is somewhat limited because space for only two digits is provided in the IDCW for the subscript.

Object String

In addition to the Name Tables, the Translator produces a single memory string called the object string, which contains the code directly executed by the Central Processor.

All language components have been translated into a post-fixed string form in the object string. All variable references are made through the data descriptors in the Name Table. The object string is not at all altered while the program is running, and consists of operands and operators. The operands are pushed onto a LIFO stack as they are encountered. When an operator is encountered, it is passed to the appropriate processor, which performs the operation on the operands on the top of the stack, and replaces them with the result (if any) of the operation.

Each word of object string may contain two machine instructions, one in each half of the word, each composed of an 8-bit operation code and a 24-bit address field. The codes E0 through EE never appear in the object string. Of the remaining 82 instructions, only six

use the address field as such: Block (90), If False Then Jump (B5), Name Table Pointer (D0), Direct Parameter (D4), Indirect Parameter (D5), and Transfer (D7). The Source Pointer (D9) instruction is generated by the Translator with an address in the address field, but since this instruction is treated as a No-op, the address field is not really required. Some operations must always appear in the same half of the word, so No-op's (00) are used to fill out the word where necessary.

Blocks, Labels and End-of-Statement

The first instruction of each block in the object string is the Block instruction (90). The block entry mechanism does not occur as a result of this instruction however, but as a result of a procedure call or ON block reference. The Block instruction is always placed in the second half of a word. The address field contains the address of the block's Name Table. The accompanying first halfword contains a No-op instruction.

Each block ends with an End Block instruction (B7). When this instruction is encountered, the block exit mechanism is invoked: The current stack is deleted and the calling block's stack becomes the new current stack. From this stack, the status register and program location counter are restored. (Generally, an End Block instruction is preceded by a Return instruction, which also invokes the block exit mechanism.) The stack for the main program is tagged so that a block exit from the main program causes a normal program completion shutdown of the Central Processor.

A Block instruction also appears in the object string at each label entry point. The IDCW for that label contains the address of this Block instruction. Whenever a Block instruction is encountered, the contents of this instruction's address field is compared to the location of the current Name Table. For a Go To within a block or for block entry, these two addresses will match, but not for a Go To across block boundaries. The Central Processor presumes that the Go To is directed towards a block which directly or indirectly called the currently active block, and performs block exits until the proper block is found. If the target of the Go To is not within one of these blocks, the main program will eventually be exited, and the Central Processor will shut down as if a normal completion had occurred.

For each semicolon or END statement, an End Statement instruction (BB), and a Source Pointer instruction (D9) are placed into the object string. The address field of the Source Pointer instruction contains the address of the last word of the source statement in the source string. The Central Processor treats this instruction as a No-op. The intended use of the Source Pointer instruction was to facilitate debugging by linking the location of an execution error to the offending source statement. The use of this facility was abandoned when software to decompile the object code directly to source code was developed, which provided precise resolution of the error location within the source statement and because there were problems inherent in the Source Pointer mechanism.[15,16] When the End Statement instruction is encountered, the stack is cleared of any remaining operands (simply by resetting the top-of-stack pointer), and user interrupts (if any) are handled.

Scalars and Structures in the Object String

The String Start codes, F0 through F5, always appear in the first byte of a word, and indicate the beginning of a scalar value in data string or numeric field format. If the value is one word long (indicated by a set, high order bit in the last byte), then the word is pushed onto the stack. Otherwise, a word beginning with an E0 and containing the address of the first word of the string is pushed onto the stack, and successive words of the object string are fetched (and discarded) until a word with a set, high order bit in the last byte is found. The String End character, F6, may appear in any byte of a word, but is not used in searching for the last word of a string.

The codes FC through FF have been described earlier in connection with initial structure values. These codes may be used to construct structure values on the stack as well. The scalar components of these structures in the object string may be arbitrarily complex expressions. Adjacent scalar components are separated by the Field Mark operator (DC). When a word beginning with one of the characters FC through FF is encountered, that word is pushed onto the stack. The expressions are evaluated just as if no structure operators had been encountered, and the result, or a link to it is left on the stack. At a later time, the Reference Processor must convert the linear structure value on the stack into tree format.

Name Table Pointer Instruction

The Name Table Pointer instruction (D0) is used for all references to variables, labels, and procedures. The address field of this instruction points to an IDCW, which is examined by the Reference Processor when this instruction is encountered. The action taken depends on what is found in the IDCW.

For variables and labels, a link word is pushed onto the stack. This word contains the pointer to the IDCW, and begins with a character that reflects the information found in the Name Table: E0 (link to simple variable), E1 (link to structure), or E4 (link to simple variable with value stored in Name Table). If the Name Table Pointer instruction is preceded by an IN instruction (B8), the link word will begin as follows: EA (IN reference to simple variable), EB (IN reference to structure), or EE (IN reference to simple variable with data stored in Name Table). For a label, the link word begins with the character E3.

A variable reference may be followed by a subscript list. Expressions to evaluate each subscript are followed by an Integerize operator (DA) or a Colon operator (BA), as described above. Following this subscript list is the Perform Subscription instruction (DD). Actual evaluation of this subscripted variable reference is deferred until the value or location absolutely must be bound to continue, at which time the Reference Processor will perform the subscription. A major change to the original design was made when problems associated with an earlier binding were encountered.[17]

Arithmetic Operations

When an arithmetic operator is encountered in the object string, the Format Processor first converts the operands to numeric field format (if necessary), and then the Arithmetic Processor (or the Format Processor for the Absolute Value, Negate, and Format operators) carries out the operation.

The Add (AB), Subtract (AD), Multiply (AA), and Divide (AF) operators cause the top two operands on the stack to be replaced by their sum, difference, product, or quotient, respectively, in numeric field format. The value is either stored directly on the stack, or a link to the temporary value is stored on the stack if the result contains more than nine significant digits.

A two digit Limit register places an upper limit on the number of significant digits to which these four operations are carried out, and hence, an upper limit on the precision of the results. (The precision of the result may be less than this limit, depending on the precision of the operands.) This register may be read or written by software, and is treated as a symbolic variable. The Limit instruction (BC) causes a word to be pushed onto the stack beginning with a BC. This word is later converted to the two digit value in data string format if the value is being read. A one bit Limited flag is set or cleared as a result of these operations depending on whether or not the precision of the result would have been more than the Limit register allowed. This flag can only be read by software. When the Limited instruction (98) is encountered, the value is pushed onto the stack as a "0" or "1" in data string format, and the flag is cleared.

There are six numeric comparison operators: Equal to (BD), Not Equal to (9D), Greater than (9E), Less than (9C), Greater than or Equal to (9B), and Less than or Equal to (9A). These operators cause the top two operands on the stack to be replaced by a "1" or a "0" (in data string format) based on the outcome of the comparison. When numbers of unequal precision are compared, the comparison is carried out to the precision of the least precise operand.

The two monadic arithmetic operators, Absolute Value (99) and Negate (DB), simply alter the sign of the top operand on the stack as required. The Format operator converts the numeric operand second from the top of the stack to data string format using a control string on the top of the stack as a template.[18]

Character String Operations

The character string operations are carried out by the Format Processor, which will also unpack numbers (if necessary) from numeric field format to data string format before proceeding.

The Join operator (8E) replaces the two string operands on the top of the stack with a string formed by concatenating the operands. The Mask operator (8F) is a general purpose string editing operator.[18] The operand on the top of the stack is used as an editing template on the second operand. The result replaces the operands on the stack.

There are three character string comparison operators: Before (88), Same (89), and After (8A). As for the arithmetic comparison operators, the two operands are replaced on the stack by a "1" or a "0" (in data string format) based on the outcome of the comparison. Two strings must be of equal length, as well as contain the same characters in the same order for the result of the Same operator to yield a "1". The Before and After operators compare two strings based on a special collating sequence (null character, special characters, AaBbCc...XxYyZz012...789) rather than on the magnitude of the internal eight bit ASCII representation as is customarily done. When comparing unequal length strings, the shorter string is considered to be padded on the end with null characters.

Boolean Operations

There are three Boolean operators: Not (8B), And (8C), and Or (8D). The operands used in Boolean expressions are character strings formed from the three characters "0", "1", and the space character (which is ignored). The Not operator replaces the top operand on the stack with a string (or link to a string) formed from the operand by converting each "0" to a "1", each "1" to a "0" and removing each space character. The And and Or operators replace the top two stack operands with their bitwise conjunction or disjunction, respectively. When these latter two operators are used on unequal length operands (excluding spaces), the shorter operand is considered to be padded on the end with "0"s.

The Format Processor is responsible for executing the Boolean operations. As for the character string operations, operands will be converted to data string format if necessary (since, for example, "100" could be both the bit string of length three, and the integer following 99).

Assignment Operations

There are two assignment operators: Left Assign (DF), and Right Assign (9F). The former assigns the value indicated by the top operand on the stack to the location indicated by the operand second to the top of the stack; the latter assigns in the opposite direction. Until one of these operators is encountered in the object string, it is not known whether the preceding operands are to be used for locations or values. This is why pointers to IDCW's are used on the stack for variables, rather than the variable's value or location. Before the assignment is carried out, any links to values are converted to actual values on the stack, even if this may require more than one word. This operation, and the final assignment of value to location are performed by the Reference Processor. For a structure assignment statement, the value appears on the stack as a structure in linear format. The Reference Processor is responsible for converting this value to tree format as it stores the value.

Transfer and Go To Instructions

The Transfer instruction (D7) simply resets the program location counter to the value in the instruction's address field. This instruction is generated to detour around code in the object string which is not to be executed in-line, such as initial data values, internal blocks, Else clauses, and code to evaluate actual parameters. It is not generated as a result of the SPL Go To statement however.

There is also a conditional transfer instruction which is generated for each SPL If statement, called the If False Then Jump instruction (B5). It is used to jump around the code for the Then clause, to the code for the Else clause (if any) if the conditional expression in the If statement is false. Preceding this instruction there will be an expression which should result in a single Boolean value on the top of the stack. (Anything else will cause a processing error shutdown.) This value is tested and if it is a "0", then the program location counter is set to the value in the instruction's address field. Otherwise, execution continues at the instruction following the If False Then Jump instruction.

A Go To instruction (95) is generated for each Go To in the source program. Unlike the two jump instructions, it contains no address in its address field. The target of the Go To is found indirectly from the top operand on the stack. This operand may be a link to a label (containing a pointer to the IDCW for a label), or a simple or subscripted variable reference. The Reference Processor is called on to evaluate the variable reference and place a label value on the stack. Recall that label values also contain pointers to the IDCW's of labels. Until the IDCW is examined, it is not known whether the label has been defined, or even if the IDCW is for a label at all. If the IDCW is not for a defined label, a processing error shutdown will result. Otherwise, the IDCW will contain the address of a word containing a Block instruction where execution will continue as described above.

Procedure Call, Parameters, and Return

The code in the object string for a procedure call (or function reference) will in general consist of three parts: a Name Table Pointer instruction for the procedure, code to evaluate any indirect parameters, and parameter instructions. The code for indirect parameters is not executed in-line, so if there are any indirect parameters, then a Transfer instruction follows the Name Table Pointer instruction directed at the first parameter instruction to be executed. Then for each actual indirect parameter is the code to evaluate that parameter, followed by a Parameter Return instruction (D8). (This instruction is of course used when the parameter is referenced to signal the end of the actual parameter code.)

Lastly, if there are any direct or indirect parameters, there are the parameter instructions, which will appear in the object string, two per word, in the order opposite to the order in which the corresponding parameters appear in the source program. For indirect parameters, there will be an Indirect Parameter instruction (D5) containing the address of the code to evaluate the actual parameter. For direct parameters, there will be a Direct Parameter instruction (D4) containing the address of the IDCW for the actual parameter.

After the Name Table Pointer instruction is encountered, the Reference Processor informs the Instruction Sequencer that the identifier is for a procedure. The Instruction Sequencer then begins looking for parameter instructions, ignoring No-ops and executing any Transfer instructions. The parameter instructions are pushed onto the stack, one per word. The first instruction which is not a No-op, Transfer, or parameter instruction will be executed on return from the procedure (the return point). The parameter instructions are then popped from the stack (note that the stack operations reverse their order); the IDCW's of the formal parameters are modified as described above. If the number of formal parameters does not equal the number of actual parameters, a processing error shutdown occurs. Following these operations, a block entry is made at the start of the procedure's object code, which is found from the procedure's IDCW.

A reference to a direct formal parameter is identical to a reference to a Global variable, label, or procedure. When an indirect formal parameter is referenced, a word is pushed on the stack containing the state of the Instruction Sequencer. Program execution then continues at the address designated in the formal parameter's IDCW, just as if no parameter reference was in progress. (The code to evaluate the

actual parameter may itself contain indirect parameter references.) When the Parameter Return instruction (D8) is encountered, the top of stack register contains the actual parameter (value or address) or a link to it. The state of the Instruction Sequencer is restored from the topmost word of the stack in memory. Program execution then continues as before the parameter reference.

The Return instruction may or may not return a value (or location), which may or may not be used. The internal top-of-stack register will contain the operand to be returned, if any. The block exit mechanism invoked by the Return instruction (D6) will delete the memory space occupied by the current stack, but will not clear this register. So this register becomes the top of stack for the calling block. If the calling block is expecting a value, and the register is empty, a processing error shutdown will result. If a value is returned, and none is required, no processing error shutdown will result. This is because the End Statement instruction, which will follow a simple procedure call, clears all operands from the stack, including the contents of the top-of-stack register.

Input and Output

Input and Output operations transfer and transform information between memory and the outside world. Six I/O status bits are maintained by the Instruction Sequencer to indicate the type and mode of the I/O operation. When the Input instruction (80) is encountered, the Input I/O status bit is set, and the remaining bits are cleared. The Output instruction (81) causes the Output I/O status bit to be set and the others to be cleared.

The remaining four bits are used to indicate the I/O mode. Following the Input or Output instruction in the object string there may be a String instruction (A3), a Data instruction (A1), or, for Input only, an Exact instruction (A4) or an Empirical instruction (A5). For each of these instructions there is a corresponding I/O status bit which is set when the instruction is encountered. The List mode is indicated by the absence of any other I/O mode.

The I/O mode determines the type of data transformation to be performed. In memory, the data may be a scalar value in data string or numeric field format, or a structure in tree format. In the outside world, the data exists as an ASCII character string. Structures in the outside world are represented explicitly using the characters "<" and ">" to delineate each structure or substructure, and the field mark character "†" is used to separate adjacent scalar components. It is the Input/Output Processor's responsibility to transform data between this explicit structure format, and the internal, linear format, if the I/O mode calls for such a transformation.

Data may be directed to or from a number of different I/O devices. If the default device, with associated device number zero, is not to be used, then following the I/O type and mode instructions , there will be code for an expression for the device number, followed by a To instruction (A0) for output, or a From instruction (B0) for input. (After the To or From instruction, a Comma instruction is expected but ignored.) The code to evaluate the expression is executed, and a value or link is left on the stack. The To or From instructions force the value to be placed on the stack, and then to be integerized, as for subscripts. The two least significant (BCD) digits are extracted and designated as the device number for the operation. The Channel Controller associates devices with device numbers.

Next to appear in the object string will be the I/O items, separated by Comma instructions (AC). The Comma instruction causes the value of the preceding I/O item to be placed on the stack for output; for input, it causes the Input/Output Processor to get an input value which is then assigned to the preceding I/O item. The last I/O item is followed by either an End Statement or an End Block instruction, each of which is treated as having been preceded by a Comma instruction. In addition, for output, these two instructions cause the Input/Output Processor to output the values on the stack, starting at the bottom and ending at the top.

For Input Data, there will be no I/O items since both variable names and values come from the outside world. Each I/O item for Output Data will be a simple variable reference (Name Table Pointer instruction). For the remaining input modes, an I/O item may be a simple or subscripted variable reference or a procedure reference (which must eventually return a simple or subscripted variable reference). For the remaining output modes, an I/O item may be any expression. The code for each I/O item is executed exactly as if no I/O instructions had been encountered.

For all output modes, the actual value of the I/O item must be placed on the stack. A scalar value in numeric field format is converted to data string format by the Format Processor. If the value is for a structure, a temporary stack is created onto which the Reference Processor places the value, converting it from tree format to linear format. The structure is then copied to the regular stack and any numeric scalar components are converted to data string format.

For Output Data, the variable's name must be placed before the value on the stack. The name is found in the one or more words preceding the variable's IDCW, a pointer to which will exist in the top of stack register as a result of the just executed Name Table Pointer instruction. If the variable is scalar valued, a word is placed before and after the value of the variable on the stack beginning with the characters F0 and F1, respectively. The Input/Output Processor converts each of these words to the field mark character "†" to delineate the value in the output.

For Input Data, the Input/Output Processor calls on the Translator to extract the variable names and values from the input character string and perform the assignment. For Input List and Input String, the Input/Output Processor will leave a value on the stack on top of a link word pushed onto the stack as a result of executing the code for the I/O item. The Reference Processor is called on to perform an assignment operation, just as if a Left Assign instruction had been encountered.

The Exact and Empirical input modes are used to convert input values to numeric field format. A temporary stack is created onto which the Input/Output Processor places the input value. The value is moved to the regular stack and the scalar value, or each scalar component, which must be a number, is converted to numeric field format. If a precision tag was given in the input value, it is used in the conversion; otherwise, the precision is determined by the input mode. The assignment operation is then carried out as for the List and String input modes.

Pause and System

The Pause instruction (96) and System instruction (97) cause the Central Processor to load the error code register with the one byte opcode and then shut down. The hardwired System Supervisor notices that the Central Processor has shut down and examines the error code register. If the instruction was Pause, then the System Supervisor deletes the process from the Central Processor's run queue. This has the effect of halting the execution of that process. A paused process is restarted when the user presses the Continue button on his terminal.[11] If the instruction was System, then the System Supervisor executes a previously defined memory string of control words. The System instruction is used in "privileged" software to modify low level system data structures which are normally maintained by hardware. Adding or deleting a process from a processor queue is a typical example of the use of the System instruction.

Logical Memory

As mentioned previously, SYMBOL differs from most von Neumann computers in that the memory structure is not organized as a contiguous set of sequentially numbered storage cells. Instead, a "logical memory" structure implemented by the Memory Controller is imposed on top of the virtual memory system. The Memory Controller takes each virtual page and divides it up into three sections. The first four words of the page are the "Page Headers", which contain pointers and status information. One of the pointers links pages together in a forward linked list; SYMBOL, nominally used three of these "Page Lists" for each terminal.[6] The first Page List was for the

user's source program, the second for user data, the stack and Name Tables, and the third for Object String. The Page Headers also indicate available space in the page by status bits, and outside the page by a Space Available List pointer which points the the next page that contains available space. The remainder of the page is divided into twenty-eight eight-word "Groups", and twenty-eight "Group Link Words". The Memory Controller then organizes these contiguous eight word "groups" into memory strings with a doubly linked list. All of the processors other than the Memory Controller then view this logical memory structure as the fundamental memory organization of the machine.

The Privileged Memory Operations

There are sixteen instructions which operate directly with memory addresses to read or alter storage. These instructions are issued by the hardware processors or by systems programs which have been translated in "privileged" mode. A request for memory to the Memory Controller consists of a 92 bit value consisting of three fields. First is a four bit field for the Page List, followed by a twenty-four bit absolute virtual address field, and a sixty-four bit data field. These fields are transmitted to the Memory Controller, which may use and modify them, returning them to the originating processor. Each memory request is also accompanied by the terminal number. Because words in memory are not necessarily contiguous, no address indexing calculations can be performed. For this reason the memory operations are of the flavor, "Here is an address, get me the data at that address and tell me what the address of the next word is." More specifically the sixteen memory operations are as follows:

Assign Group: Used to allocate a new memory string. If the transmitted address is non-zero, the Memory Controller will try to allocate a group from the same page. If no group is available on that page, an empty group will be looked for by following the Space Available List pointer to a page which has free space. If there are no pages on the Page List with space, a new page will be allocated from the system Available Page List. If the transmitted address field is zero, then the Memory Controller will allocate a group from the same Page List as specified in the page list field. If the transmitted page list field and the address field are both zero, then a new page is allocated and the first group on the page will be allocated. The returned address is the address of the first word of the assigned group.

Fetch and Follow: Returns the data at the specified address and returns the address of the following word in the string.

Fetch Reverse: Returns the preceding word in the string and its address.

Follow and Fetch: Returns the data and address of the word following the specified address.

Store and Assign: Stores the data at the indicated address and returns the address of the successor word. If no successor word exists, a new group is allocated as indicated in the Assign Group Instruction, and is linked onto the current storage string.

Store Only: Stores the data at the indicated address. The returned address is changed by adding one to the low order three bits modulo eight. (This has the effect of wrapping the address around in the group.)

Store and Insert: Stores the word in the indicated address and returns the successor address. If the transmitted address specifies the last word of a group, then a new group is allocated and inserted between the group of the transmitted address and the group which followed it.

Insert Group: A new group is allocated and inserted after the group specified by the transmitted address. The returned address is that of the first word of the new group.

Delete String: Deletes a memory string; the transmitted address must be that of the first group of the string. The associated Page List must also be supplied so that the string, when reclaimed, can be returned to the proper available space list. If the Page List supplied is that of the user data, then pointers to substructures will be looked for, and that space will be deleted also.

Delete to End of String: Obtains the address of the succeeding group and reclaims that and all following groups. The associated Page List must also be supplied so that the reclaimed part of the string can be returned to the proper available space list. If the Page List supplied is that of the user data, then pointers to substructures will be looked for, and that space deleted also.

Delete Page List: The Page List supplied will be reclaimed for the terminal on which the request was made. This operation is handled by the Memory Reclaimer.

Reclaim Group: If the transmitted address is zero, fetch the top of the terminal's garbage stack; otherwise link the group onto its page's available group list.

Fetch Direct: Used for fetching one of the terminal header registers, or any absolute core address (rather than a virtual address). The data at the real memory address is returned. The returned address is changed by adding one to the low order three bits modulo eight.

Store Direct: Stores the data at the real memory address given. The returned address is changed by adding one to the low order three bits modulo eight.

Fetch Terminal Header: Used to fetch one of the 21 header registers associated with each terminal. The Fetch Terminal Header instruction differs from the Fetch Direct instruction in that the Memory Controller automatically inserts the terminal number into the address field. This allows the address of a particular terminal header to be specified in a terminal independent manner. The address fetched is the specified physical address with the terminal number added in shifted left by three bits.

Store Terminal Header Used to store one of the 21 header registers. The address stored into is the specified physical address with the terminal number added in shifted left by three bits.

Conclusion

The SYMBOL instruction set has now been described in enough detail to show the complexities of implementing a high level instruction set in hardware. Many further details exist, but these are relatively minor, and would probably not be of interest to the reader. One of the reasons that the instruction set had not been described earlier was that the users of the machine were not supposed to need to know about the machine level instruction set, since they were to think that SPL was the language of the machine. Unfortunately, because this actually turned out to be the case, few bothered to learn the instruction set. Only by a prolonged exposure to the low level implementation details of the instruction set were the authors made fully aware of many inefficiencies which existed in the machine. The instruction set is far from ideal; program measurements have shown that the encoding used was very inefficient. Some of these inefficiencies can be excused when it is realized that SYMBOL was an experimental machine, and that the designers were limited in the effort they could spend in optimization. Nevertheless, we feel it is important that the instruction set be documented as it was implemented. This paper, used in conjunction with the previously published papers completes this documentation.

References

1. R. Rice and W. R. Smith, "SYMBOL -- A Major Departure from Classic Software Dominated von Neumann Computing Systems," *Proceedings of the AFIPS 1971 Spring Joint Computer Conference*, Montvale, N.J., pp. 575-587, AFIPS Press (1971).
2. W. R. Smith, et al., "SYMBOL -- A Large Experimental System Exploring Major Hardware Replacement of Software," *Proceedings of the AFIPS 1971 Spring Joint Computer Conference*, Montvale, N.J., pp. 601-616, AFIPS Press (1971).
3. G. D. Chesley and W. R. Smith, "The Hardware-Implemented High-Level Language for SYMBOL," *Proceedings of the AFIPS*

1971 Spring Joint Computer Conference, Montvale, N.J., pp. 563-573, AFIPS Press (1971).

4. A. C. Bradley, "An Algorithmic Description of the SYMBOL Arithmetic Processor," Report NSF-OCA-GJ33097-CL7301, Cyclone Computer Laboratory, Iowa State University, Ames, Iowa (1973). NTIS accession number PB-222 972.
5. M. C. Dakins, "Nonnumeric Processing in the SYMBOL-2R Computer System," Report NSF-OCA-GJ33097-CL7410, Cyclone Computer Laboratory, Iowa State University, Ames, Iowa (1974).
6. H. Richards, Jr. and R. J. Zingg, "The Logical Structure of the Memory Resource in the SYMBOL-2R Computer," *Proceedings of the ACM-IEEE Symposium on High-Level-Language Computer Architecture*, New York, Association for Computing Machinery (1973). NTIS accession number PB-228 118/AS.
7. R. J. Zingg and H. Richards, Jr., "SYMBOL: A System Tailored to the Structure of Data," *Proceedings of the National Electronics Conference*, Oak Brook, Illinois 27, pp. 306-311, National Electronics Conference, Inc. (1972). NTIS accession number PB-221 286.
8. H. Richards, Jr., "SYMBOL IIR Programming Language Reference Manual," Report ISU-CCL-7301, Cyclone Computer Lab., Iowa State University, Ames, Iowa (1973). NTIS accession number PB-221 378.
9. H. Richards, Jr. and C. Wright, "Introduction to the SYMBOL-2R Programming Language," *Proceedings of the ACM-IEEE Symposium on High-Level-Language Computer Architecture*, New York, Association for Computing Machinery (1973). NTIS accession number PB-228 115/AS.
10. T. A. Laliotis, "Architecture of the SYMBOL Computer System," in *High-Level Language Computer Architecture*, ed. Y. Chu, Academic Press (1975).
11. G. J. Myers, *Advances in Computer Architecture*, John Wiley & Sons (1978).
12. L. M. Alarilla, Jr., "Storage Linking Techniques for the Automatic Management of Dynamically Variable Arrays," Report ISU-CL-7403, Cyclone Computer Laboratory, Iowa State University, Ames, Iowa (1974).
13. H. Richards, Jr. and A. E. Oldehoeft, "Hardware-Software Interactions in SYMBOL-2R's Operating System," *Proceedings of the Second Annual Symposium on Computer Architecture*, NTIS accession number PB-239 220/AS (1975).
14. D. R. Ditzel, "Program Measurements on a High Level Language Computer," *Accepted for publication in Computer* (1979).
15. D. R. Ditzel, "High Level Language Debugging Tools on the SYMBOL Computer System," *1980 Workshop on High-Level Language Computer Architecture*, Fort Lauderdale, Florida (May 1980).
16. D. R. Ditzel, "Interactive Debugging Tools for a Block Structured Programming Language," Report MCS72-03642-CL7802, Cyclone Computer Laboratory, Iowa State University, Ames, Iowa (1978).
17. C. L. Smith, C. T. Wright, and R. J. Zingg , "Problems in the Push-Down Stack Approach to the Implementation of High Level Languages," *Digest of Papers, COMPCON76*, New York, pp. 96-98, IEEE (1976).
18. D. R. Ditzel, "MASK and FORMAT: Operators for Editing and Formatting," *SIGPLAN Notices* **12**(11), pp. 28-35 (November, 1977).

High Level Language Debugging Tools on the SYMBOL Computer System

David R. Ditzel†

Bell Laboratories
Computing Science Research Center
Murray Hill, New Jersey 07974

ABSTRACT

The development of debugging tools on the high level language SYMBOL computer is described. The software system developed allows a detailed interactive investigation of the dynamic and static program structure and user variables entirely at the source program level for a procedural block-structured programming language. Source statements are "de-compiled" from the object code, descriptors and hardware maintained type tags allow the unambiguous interpretation of data values. Language constructs in the SYMBOL programming language which aid in debugging are also described. Comments are made on the evaluation of the system and how the debugging environment was affected by the high level language architecture of the SYMBOL machine.

Introduction

One of the motives often suggested for High Level Language Computers has been that they make program debugging easier. The high level language SYMBOL computer system[1,2,3] provided a unique opportunity to test out this hypothesis. In this paper the state-of-the-art debugging tools developed for SYMBOL will be presented, along with a description of how and why these tools were developed. The exposition of these debugging tools is important for two reasons. First, it documents how debugging was achieved on perhaps the most advanced high level language computer yet constructed. Second, it completes documentation on what many users observed to be the most important feature of SYMBOL -- the high level language programming and debugging environment. Only by examing this user visible system software imposed on top of the SYMBOL architecture can one make a judgement on the effect the high level architecture had on the debugging environment.

Unveiled in 1971, the SYMBOL computer system had as its prime goal to demonstrate with a full-scale working computer that a procedural general-purpose programming language and a large portion of a time-shared operating system could be implemented directly in hardware. This approach was intended to show a marked improvement in computational rates over conventional systems. Almost every aspect of the system was unique, from its eight special function processors to the totally new SPL[4] programming language. While the system was capable of running totally without system software this was rarely done. System software for SYMBOL greatly contributed to the "system" interface which appeared to the user as independent of the hardware or software implementation.

Early Debugging

Without software SYMBOL provided no facilities for debugging programs with execution errors. Consequently one of the most useful programs early in the project was a traditional interactive memory dump. This fairly small program was effective for program debugging if one was familiar with the high level instruction set and data organization. Upon detection of an execution error the System Supervisor would suspend the user program, save the 24 "header" registers in a known place, and then start up a "Monitor" program on the user's terminal. From the Monitor the user could enter the dump program to look at his dead program and its source. As with most users, the first questions to be answered are why and where. The first place to look was in the AH1 header, because one of the bytes was an "Error Code Character"; this was then translated to English by looking at one of the many Engineering Reference Cards lying around. Once the nature of the error was determined the current object code address was taken from the left half of the AH2 header. By dumping five or six words of object code at this address the user could usually encounter a Source Pointer opcode, generated by the Translator at each semicolon in the source program. The address field of the Source Pointer instruction was an absolute address pointing to the corresponding line in the original source code. This entire process could be done at a terminal in less than a minute.

The Source Pointer Problem

Using the source pointers left by the Translator to find the source line was straightforward, and so was soon programmed into the terminal Monitor. Error messages were also automatically translated to a more understandable English message. While this system of tracking down the source was simple it had the proverbial "Achilles' heel" that made it untrustworthy and potentially dangerous. A program could be interrupted, the Monitor and its subsystems invoked, and then the program could be resumed at the point of interruption. If the user edited the program source and then resumed execution of the object code, the source pointers in the object code were potentially invalid. The situation is not unlike the "dangling reference" problem encountered in block structured languages.[5] In short-lived user programs this turned out occur infrequently; systems programs on the other hand might be executing the same object code for weeks while the source was being modified, allowing source and object files which differed greatly in content and date of last modification. Debugging systems programs then brought us back to square one.

†Work done at Iowa State University under NSF grant GJ30097X

More Problems

SYMBOL provided a rather inadequate hardware text editor which was limited to specially designed terminals; this lead to the implementation of software text editors. These editors initially used SPL structures (arrays) for storing and manipulating text. Since the Translator required that the source program be in a contiguous memory string, the user's source program was copied from the text structure to the memory string just prior to translation. Source pointers generated by the Translator therefore pointed to the copy, requiring a search through the structure to find the original source and its location. Both the copying and searching processes were painfully slow; eventually this type of editor was replaced with one which worked directly on memory.

Decompile

A rather unique approach was taken to solve the above problems; a program was constructed which "de-compiled" SYMBOL object code back into SPL source statements. The decompilation process was greatly facilitated because the SYMBOL instruction set was so similar to the SPL language and by the direct and simple manner in which the Translator generated object code. Decompilation was remarkably effective in re-creating source; in most instances the decompiled statement differed from the original source only in minor ways such as the number of blanks, carriage returns, the case of letters, and the omission (in the decompiled version) of redundant parentheses.

Execution Error Diagnostics

When an execution error occurred, the user process was suspended and the System Supervisor invoked the Monitor on the appropriate terminal. Use of the decompile program, coupled with a program to interpret data values on the evaluation stack, allowed excellent diagnostics to be given entirely in terms of the high level source program. On an execution error the Monitor generated the following:

1. Notification that an execution error had occurred.
2. The nature of the error (in an understandable form).
3. The statement at which the error occurred.
4. An arrow beneath the source line pointing to the particular operator or operand causing the error.
5. If the error involved a monadic operator then its operand was printed. If the error involved a dyadic operator then both operands were identified and printed.

For example, division by zero in one program generated the following diagnostic:

```
*** EXECUTION ERROR (ZERO DIVISOR - CODE 20)
    IN THE FOLLOWING STATEMENT:

m1 := qmin * (m1 * m2 / (gamma / rho) / (minterm / (1 - f)));
                                          ^

RIGHT OPERAND:
| 00 |
LEFT OPERAND:
minterm: | -5.663 |
MONITOR IS NOW IN CONTROL
?
```

At this point the Monitor waited for commands from the user. Logical choices would be to enter the text editor and correct the problem or to enter the INQUIRE subsystem for further examination.

INQUIRE

Knowing the source line and operands involved in an error is only the first step in providing good high level language debugging tools. For proper debugging we feel it is necessary to be able to examine the contents of variables, to examine the currently active calling sequence down to the source line invoking the call of each active procedure, and to examine the state of the expression evaluation stack. Such examination should be available after an execution error has occurred or at any time during the normal running of a program. This is the function of the INQUIRE subsystem. The INQUIRE subsystem is the primary means of examining the user program variables and block structure of the program. When entered, the "command environment" is set to the block that was in execution when the user program was interrupted.

INQUIRE responds to commands from the user in the following way. Entering an identifier causes the value of that identifier to be printed, providing that it is known to the command environment. Special qualifications are given to several classes of variables. An identifier that has never been referenced is tagged as "unreferenced null". Procedure names, labels and switch components are tagged only as to type. Parameters are identified and the parameter linking is followed to the calling environment to resolve the parameter. Global variables are identified and the value in the defining environment is printed. Each scalar element of a vector is printed along with its subscript list. (Successive nulls are grouped together in an attempt to save paper.) Individual elements of a vector can also be obtained. All identifiers known to a block can be obtained with the DATA command.

Identifiers from other than the current block are available as well; one of the unique features of INQUIRE is the ways in which various blocks can be traversed. For example, the value of an identifier in the block calling the block of the current command environment is obtained by preceding the identifier name with an "up-arrow" character (↑ or ^). This specifies that the identifier is to be looked for by going out one level from the command environment, according to the dynamic nesting. In a similar manner, any number of dynamically nested blocks may be traversed by preceding the identifier name with the appropriate number of up-arrows.

The static program nesting can also be used to specify a particular block. Before this can be accomplished however, a BLOCK or PROCS command must be given. The BLOCK command prints the static block structure of the program and assigns an integer value to each block. This number provides a unique naming for each block. The character ">" followed by one of these integer values will change the current command environment to the specified block number. Setting the command environment to a block which was not a member of the calling sequence allows looking at static variables but precludes obtaining the values of any formal parameters since a non-active procedure has no calling point for parameter linkage. Selection of a non-active block also prohibits use of the up-arrow commands. In addition to printing the block structure of a program, the BLOCK command prints the names of all identifiers used in each block and an

abbreviated tag as to their data type, e.g., scalar, structure, label, procedure, etc. The PROCS command is similar to the BLOCK command with the exception that it prints only the block structure.

The WHERE command locates the statement in execution, prints it on the console device, and then pauses. Pressing the Continue button will cause one succeeding statement to be printed before pausing again. This sequence is exited by pressing the F0 special function button. The most useful use of the WHERE command is in conjunction with the "up-arrow" feature described previously. A command consisting of ↑WHERE prints the statement that called the procedure in execution, and thereby reveals the name of the procedure and its actual parameters. In this manner the calling sequence may be examined any number of levels on a very specific basis. Since an entire statement is printed using the ↑WHERE command, a more specific reference is needed to isolate the exact point of execution. A large expression may contain many operators and operands; for example, the statement in the diagnostic of the previous section contained several division operations. To isolate the exact point of execution or error a pointer is printed beneath the statement directly below the appropriate operand or operator.

Examination of expressions which may have been partially evaluated is possible using the STACK command. This command prints the top entry of the stack and then pauses. Pressing Continue prints one successive stack entry and then pauses again if the bottom of stack has not been reached. Pressing the F0 button before the bottom of stack is reached will cause a return to the command mode. As SPL is a block-structured language, there is a separate stack associated with each active block. The stacks of other active procedures are accessed by preceding the STACK command with the desired number of up-arrows or by first entering the appropriate block via the ">" command.

If a program was interrupted by pressing the interrupt key, the program may be resumed at the point of interruption by using the RESUME command or at a label by using the GO TO command. The GO TO command has the restriction that the label must be in a block which is currently active. The RESUME command may not be used after an execution error although GO TO may be used regardless of the cause of the interrupt.

If an identifier has an ON block associated with it, that ON block may be enabled or disabled from INQUIRE. A more detailed description of ON blocks follows.

A brief description of INQUIRE commands is available from the terminal with the HELP command. A listing of the HELP text is given in Appendix 1. Appendix 2 shows a sample terminal session using INQUIRE.

ON Blocks

ON blocks are an SPL language construct extremely useful for debugging. An ON block is similar to a procedure, in that it is a series of statements invoked from some calling point. Unlike procedures, however, invocation of an ON block is caused by the occurrence of an implicit event specified by a list of names following the ON declaration. If the list contains a variable name, the ON block will be invoked immediately after an assignment to that variable occurs. If the list contains a label, the ON block will be invoked upon encountering a GO TO statement to that label before the transfer actually takes place. If the list contains a procedure name, the ON block will be invoked upon encountering a call to that procedure before entry to the procedure takes place. If the list contains the word INTERRUPT, the ON block will be invoked when the user presses one of the function buttons (F1 thru F15).

The ON block facility bears a resemblance to the PL/I ON CHECK condition. The major difference is that in SPL multiple ON blocks are allowed to exist within a particular environment (scope) and that the invocation of ON blocks can be controlled selectively for individual identifiers. The IBM PL/I(F) compiler makes no provision for dynamically enabling or disabling the CHECK condition, and while the ON CHECK units may be dynamically switched around, such switching applies equally to all variables to which the CHECK condition applies. In SYMBOL invocation of an ON block for a particular identifier is controllable by the SPL ENABLE and DISABLE statements.

A typical use of an ON block is shown in Figure 1, which illustrates a method to discover where a variable is assigned undesired (or desired) values. The value of I will be printed every time it is modified and the user can then decide whether to continue, or interrupt his program and diagnose further with INQUIRE. Once the user is satisfied that the particular portion of the program being monitored by an ON block is behaving properly the ON block can be disabled from INQUIRE. The implementation is a major advance over what is possible in most systems in that no extra code needs to be generated to invoke an ON block nor does the program have to be re-compiled to turn on or off the invocation of an on block. This has major benefits in terms of execution efficiency and the ability to debug non-stop programs, not to speak of the time saved in editing and re-compiling programs after changing the debugging options. The ON block is also a clean way of debugging a program in that it concentrates the debugging code in one place, in contrast to spreading debug I/O throughout a program; this practically eliminates needing to "clean up" a program after debugging.

```
ON I;       NOTE  This block invoked whenever I is assigned to;
   GLOBAL I;
   OUTPUT |The value of I is|, I;
   PAUSE;
END
```

Figure 1. Simple ON block

The descriptor orientation of SYMBOL was a major factor in the efficient implementation of the ON block facility. Descriptors were sixty-four bits long and contained sixteen tag bits and two twenty-four bit address fields. An identifier with an associated ON block had the left address field pointing to the identifier value and the right address field pointing to the start of the object code of the ON block. The ENABLE and DISABLE statements either set or reset and "ON Enabled" bit in the tag field. As the descriptor had to be referenced for every identifier reference, checking to see if an identifier had an ON block associated with it could be done in parallel with normal accessing without loss of performance.

Evaluation

To a large extent the tools developed show what was easy or reasonable to do with the SYMBOL architecture. Descriptors and type tags allowed the type and values of data objects to be easily interpreted. The additional level of indirection imposed by descriptors was extremely important in implementing ON blocks. Being able to selectively Enable or Disable ON blocks from INQUIRE or dynamically in the users program without recompilation drastically reduced the compilations and editing that might otherwise have been required. Some credit has to be given to the designers of the SYMBOL language for introducing ON blocks with Enable and Disable statements.

Decompilation is a subject which requires several comments. First, it must be realized that we had almost no control over the instruction set or the code generated by the Translator. While we could have generated better code with a software compiler, experimentation proved a software compiler to be to be impractical because of its slow speed. Fortunately, the high level instruction set and simple code generation algorithms made object code relatively easy to invert. On the negative side, decompilation was not trivial (some 900 lines of code), nor was it fast (3 to 10 seconds/statement). Decompilation has several other negative characteristics. Starting to decompile from the middle of control flow instructions (eg. if-then-else, looping, procedure body) made decompiling the bottom part of the flow syntax difficult; this could have been much easier if, for example, the jump over an "else" clause had been distinct from other jumps. Comments and declarations generated no code, and hence would never re-appear in a decompiled program. The minor differences in number of blanks, carriage returns, and case of letters were very irritating when trying to find the "same" source line in an editor by using an exact string search. On the whole, if one has control over the compiler there exist much better techniques for mapping object code back into source statements.[6,7] Decompilation was used in our case because we had few other options.

Conclusion

Users of the SYMBOL system were very pleased with the programming and debugging environment; in particular with the way INQUIRE allowed the investigation of their block structured programs. The software debugging tools were the finishing touch in making SYMBOL a High Level Language Computer System,[6] rather than just a machine with a fancier instruction set. The disappointing part for ex-users of the SYMBOL system is that there are no inherent reasons why similar features could not be provided even on low level language machines, yet such debugging systems are not appearing. What the SYMBOL architecture did for us was make the job of building some of our tools easier than would have been possible on a more traditional machine.

Acknowledgements

The author wishes to acknowledge the assistance and dedication of R. Cmelik, P. Hutchison, W. Kwinn, H. Richards, and R. Wolf, programmers who contributed to the software development of this very useful computer system. S. R. Bourne and S. C. Johnson are thanked for their comments on this manuscript. Financial support for the SYMBOL Project at Iowa State University was been provided by the National Science Foundation under grant GJ33097X and by the Iowa State University Engineering Research Institute. Preparation and typesetting of this paper was aided immeasurably by the text preparation facilities on the UNIX† system.

References

1. R. Rice and W. R. Smith, "SYMBOL -- A Major Departure from Classic Software Dominated von Neumann Computing Systems," *Proceedings of the AFIPS 1971 Spring Joint Computer Conference*, Montvale, N.J., pp. 575-587, AFIPS Press (1971).
2. W.R. Smith et al., "SYMBOL -- A Large Experimental System Exploring Major Hardware Replacement of Software," *Proceedings of the AFIPS 1971 Spring Joint Computer Conference*, Montvale, N.J., pp. 601-616, AFIPS Press (1971).
3. G. D. Chesley and W. R. Smith, "The Hardware-Implemented High-Level Language for SYMBOL," *Proceedings of the AFIPS 1971 Spring Joint Computer Conference*, Montvale, N.J., pp. 563-573, AFIPS Press (1971).
4. H. Richards, Jr., "SYMBOL IIR Programming Language Reference Manual," Report ISU-CCL-7301, Cyclone Computer Lab., Iowa State University, Ames, Iowa (1973).
5. L. M. Chirica, T. A. Dreisbach, D. F. Martin, J. G. Peetz, and A. Sorkin, "Two Parallel EULER Run Time Models: The Dangling Reference, Imposter Environment, and Label Problems," *Proceedings of the ACM-IEEE Symposium on High-Level-Language Computer Architecture*, College Park, Maryland.
6. D. R. Ditzel and D. A. Patterson, "Retrospective on High-Level Language Computer Architecture," *Proc. of 7th Ann. Symp. on Computer Architecture*, La Baule, France (May 1980).
7. H. P. Katseff, "Symbol Table Format for Sdb," Internal Technical Memorandum, Bell Laboratories, Holmdel, New Jersey (July 1979).

† UNIX is a Trademark of Bell Laboratories

Appendix 1. INQUIRE Help Text

This is the Inquiry subsystem, which permits examination of user-program variables and block structure. The following inputs are accepted:

1. An identifier. The value of the identifier will be printed, if possible. Otherwise an appropriate message will be produced. "Identifier" here includes LIMIT and LIMITED.
2. "LIMIT=n" where n is a number between 0 and 99. The value of LIMIT is set accordingly.
3. The character ">" followed by:
   a. a number obtained from the output produced by the /BLOCK command (see below),
   b. a string of one or more "<" characters, or
   c. nothing.

   This respecifies the command environment as:

   in case a. the specified block,

   in case b. one block out from the current setting for each "<" in the string (following the dynamic nesting, i.e., the order of activation),

   in case c. the environment which was current when the Monitor was invoked.

   When the Inquiry mode is entered, case c. is assumed. In case b. if the current command environment was not active when the Monitor was invoked, a message is printed and the command environment is not changed.
4. The character "/" followed by a command keyword. Only enough of the keyword to distinguish it from all others is required. The keywords are described in the following paragraphs:
5. "/BLOCK". The static block structure of the user program is printed. each block is identified to the extent possible, the names and attributes of all identifiers known in each block are listed, and each block is assigned a reference number for use in setting the command environment (see paragraph 3). The current command environment and the blocks which were active when the Monitor was invoked are identified. The listing may be terminated by pressing F0.
6. "/DATA". The values of all identifiers known in the current command environment are printed, similarly to paragraph 1. To cancel, press F0.
7. "/WHERE". If the specified block was active when the Monitor was invoked, a reconstruction of the statement which was being executed will be displayed, and the Monitor will pause. Pressing CONTINUE will evoke consecutive statements; pressing F0 will direct the Monitor to input a new Inquiry command.
8. "/STACK". If the specified block was active when the Monitor was invoked, the top item on its stack will be displayed similarly to paragraph 1. Pressing CONTINUE will display successive stack items; pressing F0 will direct the Monitor to input a new Inquiry command.
9. "/ENABLE". An identifier is requested, and the ON-block associated with the identifier is enabled. If the identifier does not have an ON-block, an appropriate message is produced.
10. "/DISABLE". Behaves similarly to paragraph 9, but the ON-block is disabled.
11. "/GO TO". A label is requested, and user program execution is resumed at that point. The label must be in an environment which was active when the Monitor was invoked.
12. "/MONITOR". Return to Monitor.
13. "/EDIT". Equivalent to "/MONITOR" followed by "EDIT".
14. "/RESUME". Equivalent to "/MONITOR" followed by "RESUME".
15. "/PROCS". Similar to "/BLOCK", but does not list the identifiers in each block.

Any of the above inputs may be prefixed by one or more "<" characters. This will cause the command environment to be respecified as in paragraph 3b, but for that one input only.

If F0 is pressed while in input, the input is ignored.

Appendix 2. Example Program and Execution

```
NOTE Demonstration program. Keywords capitalized. User input italicized;

Inum|1|; Jnum|2|; Console|1|; NOTE Initial value statements;
Vector << 1|2|3 > < One|Two|Three > < 123,456.78|3×3 Matrix|>>;
Vector[345,6] ← |A Scalar string.|;

Repeat Scan:
    OUTPUT|What is LineA ?|; INPUT LineA;
    OUTPUT|What is LineB ?|; INPUT LineB;
    perform lexical scan( LineA, LineB, Stmnt );
Until( Inum EQUALS Jnum , Repeat Scan );

PROCEDURE WhichRoutine( name );
    GLOBAL Inum, Jnum;
    IF      name SAME |PARSE|
    THEN RETURN 1;
    ELSE IF name AFTER |M|
           THEN Inum ← 20; RETURN 2;
           ELSE Jnum ← 3; RETURN 3;
    END   END
END

PROCEDURE Perform lexical scan( String1, String2, Statement );
    SWITCH Routine< Routine1|Routine2|Routine3 >;

    S1 ← NoBlanks( String1 );
    target ← WhichRoutine( String2 );

    GO TO Routine[ target ];

Routine1:       Statement ← ( S1 FORM. Γ|$DDD.IFD|MASK|4SA.FC|) JOIN|Long|;
                RETURN;

Routine2:       Statement ← test( String1 BEFORE String2 AND Target EQUALS 2 );
                RETURN;

                PROCEDURE test( boolop );
                    OUTPUT|Paused.|;
                    PAUSE;
                    IF boolop THEN RETURN 0 ELSE RETURN 1 END
                END

                PROCEDURE NoBlanks( line );
                    BLOCK
                        test ← 5;
                    END
                    RETURN line MASK|FA |;
                END
END NOTE End of Perform lexical scan;

PROCEDURE Until( Condition , Label );
    IF Condition THEN RETURN ELSE GO TO Label END
END

ON Jnum;
    GLOBAL Jnum, Inum, Console;
    IF      Jnum EQUALS Inum OR Inum GREATER THAN 17
    THEN OUTPUT TO Console,|Error Detected – Jnum Invalid|,|Paused.|;
           PAUSE;
    END
END
```

Input from user is italicized.
Comments are in boldface.

run

What is LineA ?
493.08
What is LineB ?
TZB123
Paused.

(At this point the user presses the interrupt key.)

MONITOR IS NOW IN CONTROL.

?inquire

!where
PAUSE ;
↑
IF boolop THEN
RETURN 0;

!!where
Routine2: Statement ← test(String1 BEFORE String2 AND target EQUAL 2);
↑

!procs
MAIN PROGRAM = 1 ACTIVE (LEVEL 1)
ON BLOCK FOR Jnum = 2
PROCEDURE Until = 3
PROCEDURE perform lexical scan = 4 ACTIVE (LEVEL 2)
PROCEDURE NoBlanks = 5
INNER BLOCK = 6
PROCEDURE test = 7 ACTIVE (LEVEL 3)
INTERRUPT IN THIS BLOCK
CURRENT COMMAND ENVIRONMENT
PROCEDURE WhichRoutine = 8

string1
"STRING1" IS NOT KNOWN IN THE SPECIFIED ENVIRONMENT.

!string1
STRING1: formal parameter.
The actual parameter is "LineA" (in the calling environment).
LineA: |493.08|

!statement
STATEMENT: formal parameter.
The actual parameter is "Stmnt" (in the calling environment).
Stmnt: null

/resume

What is LineA ?
555.937
What is LineB ?
A1B2C3.
Error Detected – Jnum Invalid
Paused.

(User presses interrupt.)

MONITOR IS NOW IN CONTROL.
?inquire
/where
PAUSE ;
↑

!!where
Jnum ← 3;
↑ **(User presses Continue for another source line.)**
RETURN 3;

!!!where
target ← WhichRoutine(String2);
↑

```
!!!!/where
perform lexical scan(LineA, LineB, Stmnt);

!!!!/where
ATTEMPT TO BACK UP BEYOND OUTERMOST BLOCK

/procs
MAIN PROGRAM = 1  ACTIVE (LEVEL 1)
    ON BLOCK FOR Jnum = 2  ACTIVE (LEVEL 4)
  **INTERRUPT IN THIS BLOCK**
  **CURRENT COMMAND ENVIRONMENT**
    PROCEDURE Until = 3
    PROCEDURE perform lexical scan = 4  ACTIVE (LEVEL 2)
        PROCEDURE NoBlanks = 5
            INNER BLOCK = 6
        PROCEDURE test = 7
    PROCEDURE WhichRoutine = 8  ACTIVE (LEVEL 3)

/resume
***     EXECUTION ERROR ("GO" TO NON-LABEL. - CODE 82)
        IN THE FOLLOWING STATEMENT.

GO TO Routine[target];
     ↑

OPERAND:
null

MONITOR IS NOW IN CONTROL.

?inquire
target
TARGET: [3]

routine[3]
ROUTINE[3]:  "Routine3" (switch component)

routine3
ROUTINE3:  null

/edit
?search FOR "Routine2 " FROM LINE 1
  36 : Routine2:  Statement - test( String1 BEFORE String2 AND Target EQUALS 2 ),

?insert AFTER LINE 36
2

routine3.  NOTE adding this code after discovery of missing label, RETURN.
?inquire
/go TO:  LABEL - Routine1

What is LineA ?
999.089
What is LineB ?
    (User presses interrupt.)

MONITOR IS NOW IN CONTROL.

?inquire
/block
MAIN PROGRAM = 1  ACTIVE (LEVEL 1)
**INTERRUPT IN THIS BLOCK**
**CURRENT COMMAND ENVIRONMENT**
Inum, Jnum(ON), Console, Vector(S), Repeat Scan(1), LineA, LineB,
perform lexical scan(Pr), Stmnt, Until(Pr), WhichRoutine(Pr)

        ON BLOCK FOR Jnum = 2
        Jnum(G,ON), Inum(G), Console(G)

        PROCEDURE Until = 3
        Condition(Pa), Label(Pa)
```

Explanation of tags		
G	-	Global
L	-	Label
ON	-	Identifier has ON block
Pa	-	Parameter
Pr	-	Procedure
S	-	Structure

```
PROCEDURE perform lexical scan = 4
String1(Pa), String2(Pa), Statement(Pa), Routine(S), Routine1(L),
Routine2(L), Routine3, S1, NoBlanks(Pr), target, WhichRoutine(G,Pr),
test(Pr)

    PROCEDURE NoBlanks = 5
    line(Pa)

        INNER BLOCK = 6
        test

    PROCEDURE test = 7
    boolop(Pa)

PROCEDURE WhichRoutine = 8
name(Pa), Inum(G), Jnum(G,ON)
```

test
"TEST" IS NOT KNOWN IN THE SPECIFIED ENVIRONMENT.

>4
test
TEST: procedure.

>6
test
TEST: |5|

>8
/data
name: formal parameter.

Inum: global. In the defining environment,
Inum: |20|

Jnum: global. In the defining environment,
Jnum: |3|

>1
/data
Inum: |20|

Jnum: |3|

Console: |1|

Vector
    [1,1]: |1| [1,2]: |2| [1,3]: |3|
    [2,1]: |One| [2,2]: |Two| [2,3]: |Three|
    [3,1]: |123,456.78| [3,2]: |3×3 Matrix| [3,3]: | |
    [4-344]: nulls
    [345,1-3]: nulls [345,6]: |A Scalar string.|

Repeat Scan: label.

LineA: |999.089|

LineB: |A1B2C3.|

perform lexical scan: procedure.

Stmnt: |$555000021.xmg|

Until: procedure.

WhichRoutine: procedure.

/disable: IDENTIFIER = *jnum*
/enable: IDENTIFIER = *repeat scan*
NO ON BLOCK FOR REPEAT SCAN

/monitor
?terminate

END OF TERMINAL SESSION.
PROCESSING TIME: 24.2 SEC.
DURATION OF SESSION: 60.2 MIN.

PRESS CTRL-Q TO START